平成18年度 修士論文

イーサネットのみを利用した
組込み計算機用カーネル開発システムの
設計と実装

電気通信大学 大学院情報システム学研究科

情報システム設計学専攻

0550029　濱 善幸

指導教員　多田 好克　教授
前川 守　　教授
大森 匡　　助教授

提出日　平成19年1月30日

目 次

図 目 次

表 目 次

第 1 章

背景と目的

1.1 背景

近年組込み計算機と呼ばれる、小型で低スペックではあるが、低コストな計算機の利用が進んでいる。表 1.1 に示すように、OS を Linux に限定した場合、全世界における組込み計算機及び関連するソフトウェア等の市場規模は 2000 年には 28.2 百万ドルであったが 2005 年には 306.6 百万ドルにまで増加しており、成長は年々加速している [1]。また 2005 年には情報処理学会に組込みシステム研究会が誕生するなど、活発な研究・開発が望まれる分野となっている。

表 1.1: 世界における組込み計算機等の市場規模 (組込み Linux 入門 [1] より引用)

年	2000	2001	2002	2003	2004	2005
金額 (百万ドル)	28.2	55.2	89.8	140.5	213.3	306.6

組込み計算機が普及した理由としては、携帯電話や情報家電など様々な用途があること、それらの用途の市場が拡大している事が挙げられる。表 1.1 の 2000 年と 2005 年の市場規模内訳を表 1.2 に示すが、通信機器、通信インフラといったテレコム/データコム分野と、携帯電話、PDA、デジタルテレビなどをはじめとするデジタルコンシューマ機器の 2 つの分野での出荷数が特に増加している。また、他の分野でも着実に出荷が増えている。

表 1.2: 世界での分野別の組込み Linux 出荷数 (単位は百万ドル) (組込み Linux 入門 [1] より引用)

分野	2000 年	2005 年
テレコム/データコム	13.2	126.7
コンシューマエレクトロニクス	8.9	121.1
FA(ファクトリーオートメーション)	1.4	14.2
リテイルオートメーション	1.0	8.7
OA(オフィスオートメーション)	0.9	6.6
軍事/航空	0.9	9.4
自動車	0.7	7.4
ストレージ/サーバ	0.6	6.6
医療	0.4	4.0

ここまで組込み計算機が普及した理由としては、ハードウェアの高性能化・低価格が実現し様々な用途に使えるようになったこと、OSやその上で動作するアプリケーションなど既存のソフトウェア資産の流用による開発期間の短縮や低価格化が可能であること、が挙げられる。

組込み計算機の用途拡大やハードウェアの高性能化は、組込み計算機を対象としたソフトウェア開発にも大きな変化をもたらした。

昔はハードウェアを限界まで使うため、アセンブラで時間をかけて簡単なモニタ上にコーディングされていた。しかし現在はなどの言語を使ってLinuxなどのOSを動かし、その上で動作するソフトウェアを開発するのが主流となってきている。

組込み計算機で使用されるUNIX系OSとしてはLinuxが有名である。Linuxを選択する理由としては、次のような点が挙げられる。

- ソースが入手可能
- ロイヤリティフリー
- デバイスドライバが豊富
- ネットワークプロトコルやミドルウェアが豊富
- 豊富な運用実績

組込み計算機を対象とするソフトウェアの開発例として、街頭に設置する情報検索端末のソフトウェア開発を行う場合を考えると、タッチパネルを動作させるためのドライバソフトウェアや情報検索を行うソフトウェアを開発する必要がある。また、CDのバーコードを読み取りそのCDの楽曲を再生する端末のソフトウェアを開発する場合には、バーコードリーダを動作させるためのドライバソフトウェアや対応する楽曲を検索するソフトウェアの開発をしなければならない[2]。このように、組込み計算機を対象とするソフトウェアの開発では、ドライバソフトウェアなど低いレイヤの開発を伴なう事が大きな特徴となっている。

組込み計算機は小型で低コストという特徴がある。そのため、キーボード端子やビデオ出力端子を持たず、直接開発を行うことはできない。そこで開発対象の組込み計算機 (以下 target と呼ぶ) とは別の、ハードウェア資源も潤沢な開発用計算機 (以下 host と呼ぶ) でソフトウェアを開発し、ネットワーク経由で target に転送し動かす、クロス開発と呼ばれる手法が取られている。

デバイスドライバの異常はカーネルの異常動作に繋がる [3] [4]。デバイスドライバはカーネル空間で動作するからである。そのため、デバイスドライバなどの開発において、その動作がおかしい場合には、カーネルレベルでデバッグを行うことが必要である。

カーネルデバッガには様々な種類があるが、kgdb を用いるのが主流となっている [5]。kgdb は、target の Linux カーネル内に kgdb を組み込んで host 上で動作する gdb からシリアル端子で通信する。これにより、target 上の Linux カーネルを C 言語ソースレベルでリモートデバッグすることを可能としている。他にも充実した機能を有しているため、kgdb がカーネルデバッガとして普及しており、カーネルレベルの開発において使用されている。

ここでシリアル端子とは RS-232C とも呼ばれるインタフェースで、計算機やモデム間の接続のために用いられてきた。ノイズに強くケーブル長 15 メートル程度までデータ通信可能で、信号の検出も容易である。そのため、開発において使用される開発ボードはデバッグなどの用途で現在も利用されている。

しかし、近年更なる小型化、低コスト化の要請がある。また通信速度の遅さから通信用途としてはイーサネットが組込み計算機でも普及しており、シリアル端子が製品版からは省略されている場合が増えている。

このような製品についてカーネルレベルの開発を行いたい場合、今までのクロス開発環境を使うことができず、問題となっている。

1.2 目的

本研究では、イーサネットのみを用いて既存のコンソールやカーネルデバッグといった開発手法を実現するシステムの提案を行う。

本システムを用いた場合、既存の開発手法を用いることが出来るため、新たなソフトウェアの導入や新たな開発手法を取得する、といった開発コストの追加は発生しない。本システムは拡張性があるため、コンソールやカーネルデバッグ以外の新たな用途が出来た場合であっても対応することが可能である。詳しくは後述するが、拡張を容易にするためのインタフェースを導入したからである。また、イーサネットによるパケット通信を行うことにしたため、コンソールやデバッガなど、複数のアプリケーションを同時に使用することが可能である。

本システムではシリアル端子に代わりイーサネットを使うことにした。組込み計算機の用途として情報家電がある。情報家電では、TCP,UDP/IP ネットワークにより通信されることが主流となっている。この通信を行うためのインタフェースとしてはイーサネットが使われており、普及しているからである。

他にも表1.3に示す通り、USB など他の端子と比較した場合、通信速度は劣っているが十分な速度を備えている。また今回は 1 対 1 の通信のみ対応の実装としているが、将来的に 1 対多の通信への拡張することも予定している。開発は複数人で行われる場合もあるからである。USB などは複数人との通信に利用されるインタフェースではなく、これらを用いてシステムを構成すると、追加コストが発生する。このような理由から、イーサネットを用いた。

表 1.3: 通信端子の比較

名称	普及具合	通信速度	1 対多の通信
イーサネット	○	100Mbps	○
USB2.0	○	480Mbps	△
IEEE1394	△	400Mbps	△

第 2 章

システム概要

本章では、システムの構成やシステムの処理の流れなどについて詳説する。

2.1 システム全体像

本システムは、シリアル端子を持たない組込み計算機での開発に利用することを目的に、シリアル端子経由で行われていた通信をイーサネット経由の通信に置き換えるものである。具体的にはカーネルデバッガとコンソールのシリアル端子を経由した通信をイーサネット経由に置き換える。

そこで本節では、カーネルデバッグやコンソールの概要、また、これらの処理に通信がどのように関わるのかを述べる。

2.1.1 カーネルデバッグ

カーネルデバッグとは、カーネル内のバグの発見やその修正を行うことである。また、カーネルデバッグを行うための支援用アプリケーションをカーネルデバッガと呼ぶ。カーネルデバッガにより行えることは多種多様であるが、プログラムのスタックを表示するバックトレースやメモリの内容の表示などを行うことができる。

カーネルデバッグは、図2.1のように、デバッグ対象のカーネルとカーネルデバッガの動く開発対象 target 計算機と、カーネルデバッガの操作等を行うデバッガの動く開発用 host 計算機の、2 台の計算機にまたがって行う。このように 2 台に分

けることで、開発者は安定した host 上で target のデバッグを行うことができるようになる。

計算機を 2 台使うため計算機間の通信を行う必要があるが、この通信経路には通常シリアル端子が用いられている。

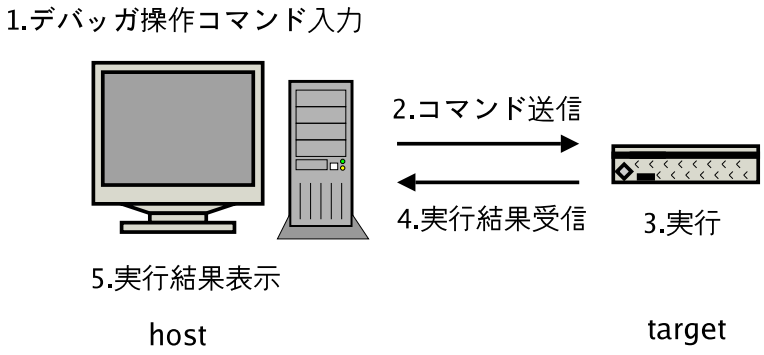

図 2.1: カーネルデバッグ実施の流れ

2.1.2 コンソール

コンソールとは、計算機を操作するために使用する入出力装置である。具体的には、計算機で実行するコマンドの入力やその実行結果の出力などに用いられ、開発時にも使用される。一般の計算機ではビデオ表示される仮想端末でコンソールを実現していることが多い。

しかし、target である組込み計算機は省スペース化や低コスト化の要求から、ビデオ表示やキーボードなどの入出力端子を持たないことが多い。そのため、host 上の入出力端子を利用してコンソール機能を実現している。具体的には、図 2.2 に示すように、target で実行させるコマンドを host から入力し、このコマンドが target に送信される。target はこのコマンドを実行し、その結果を host に向けて送信する。host は受信した実行結果の表示を行う。

この処理を行うためには、host と target 間のデータ通信が必要であるが、この

通信には通常シリアル端子が用いられている。

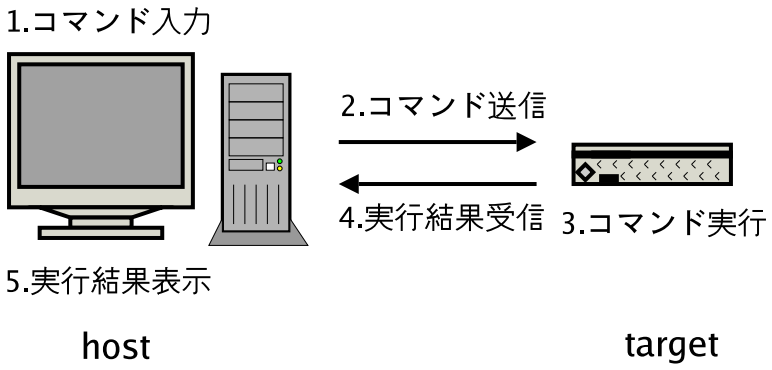

図 2.2: コンソール実施の流れ

図 2.12.2 に示すように、カーネルデバッガとコンソールは似た処理手続きを取る。そのため、1 つのシステムで置き換えが可能となる。

2.2 前提条件

本システムを動作させる環境の前提条件を以下のように定める。

- 開発用 host 計算機と開発対象 target 計算機が 1 対 1 な関係であること

  今回は実装を簡単にするため、1 対 1 の通信のみを考えることにした。

- UDP/IP が利用できるネットワーク接続がされていること。このネットワークは、安全であり回線にも十分な余裕があるものとする。

  UDP を条件としたのは、本システムの構成要素である netpoll(3 章参照) と呼ばれるネットワークモジュールが UDP にのみ対応しているため、また、UDP はコネクションレスの通信であり実装が容易となるからである。

- target の OS は Linux であり、kgdb と netpoll が動作するバージョンであること。

Linuxを対象OSとするのは、組込み計算機で使用されるOSとして普及しており、ソースが公開されていることから開発が容易なためである。

target上で起動するカーネルデバッガには広く利用されているkgdbを用いることにした。詳細は3章に示すが、targetの通信にはnetpollを使用した。そのため、この2つが動作するバージョンである必要がある。

なお、hostについてはgdbが動けばNetBSDなどの他のUnix系OSでもよい。デバッグ時のhostの役割は、コマンド入力と結果の表示のみのためであり、これらは他のUnix系OSでも可能だからである。

2.3 システム構成

本システムは2.1節に示すように、コンソールからの操作やカーネルデバッグにおいてシリアル端子を経由して行われる通信をイーサネットを経由した通信に置き換えることを目的とする。この目的を達成するため、コンソールからの操作やカーネルデバッグを行うためのアプリケーションがシリアル端子経由で行っていた通信を、イーサネット経由の通信に置き換える。

このような処理を行うためのシステム構成図を図2.3に示す。図2.3のうちnetcatとmingettyがコンソール用途のため、gdbとkgdbがカーネルデバッグ用途のためのアプリケーションである。これらのアプリケーションの説明は2.3.3節と2.3.4節にて行う。今回はこのアプリケーション間の通信を、netcontroller_io、netpoll、netpoll_connector、appli_connectorを用いて行うように設計した。

なお、hostとtarget間はUDP/IPによる通信を行う。

2.3.1 targetのシステム構成

まずtarget上の構成について考える。mingettyやkgdbはデバイスを介して処理を行うアプリケーションであり、直接UDP/IPパケットの送受信を行うことがで

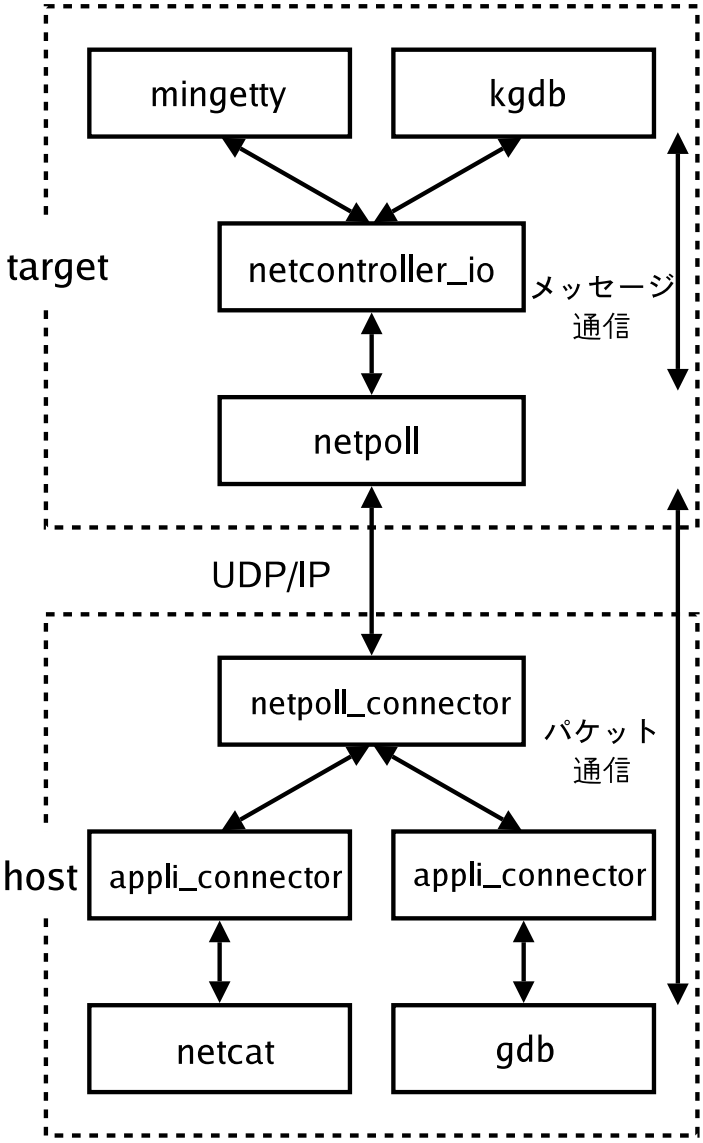
mingetty
kgdb
target
netcontroller_io
メッセージ
通信
netpoll
UDP/IP
netpoll_connector
パケット
通信
host
appli_connector
appli_connector
netcat
gdb

図 2.3: システム構成

きない。デバイスを介して処理を行うために用いられるのが、netcontroller_io である。netcontroller_io は mingetty や kgdb からの要求を受けてメッセージ通信を行い、netpoll を用いてパケット通信を行う。

target にてパケット通信を行うための機構として netpoll を用いた。

netcontroller_io は、通信先の変更に手間がかかる、待ち受け port は 1 つだけである、などの問題を有する netpoll を使わなくても、通常のネットワークスタックを用いたパケット通信を行うことが可能である。それでも netpoll を用いたのは、target のネットワーク周りが不安定であっても通信を行うことができる、というメリットがあるためである。なお詳細は 3 章で述べる。

2.3.2 host のシステム構成

host では、netcat や gdb などのアプリケーションと target 間のパケット通信を中継する appli_connector や netpoll_connector を作成した。また、1 本の経路で複数のアプリケーションからのパケットが通信される場合における、パケット識別方法を設計した。

appli_connector

appli_connector はアプリケーションから受け取ったパケットの宛先 port 番号を変更するアプリケーションである

netcat や gdb は、mingetty や kgdb とは異なり直接パケットの送受信を行うことができる。このパケットの送信について、netcat や gdb は当てはまらないが、アプリケーションの中には特定の port にしかパケット送信できない実装のものもある。詳しくは 3 章で述べるが、netpoll は複数起動することができず、target でパケットの送受信を行う port を増やすことによる解決を図ることができない。

そこで、host 上のアプリケーションの送信するパケットが target の netpoll に届く前に、パケットの送信先 port 番号を netpoll の port 番号に変更する処理を行う。

具体的には、host の内部にアプリケーションの送信するパケットの送信先port 番号を変更する機能を持った appli_connector を設けることにした。

netpoll_connector

appli_connector があれば、全てのアプリケーションが送信するパケットの送信先 port 番号を netpoll の port 番号に変更することが可能であるため、図 2.3 の netpoll_connector は不要になる。しかし netpoll には、複数起動できないという問題の他にも、送信先 IP の変更には初期化が必要という問題がある。これは、netpoll は例外的な通信を行うため、簡便な変更手段が用意されていないからである。

システム利用状況としては 1 対 1 の通信を考えているため本来不要であるが、今後の予定として 1 対多の通信に対応することを考えている。そこで、netpoll とのパケット送受信を引き受ける netpoll_connector を設けることにした。

パケット 識別方法

図 2.3 のような設計の場合、図 2.4 の UDP ヘッダ内にあるパケット送信先 port 番号や発信元 port 番号は常に netpoll や netpoll_connector の port 番号となるため、port 番号でどのアプリケーション宛てのパケットなのかを判別することは不可能である。そのため、判別するためのタグを新たに設ける必要がある。

図 2.4: UDP/IP パケット

そこで、図 2.5 のようにパケット内のメッセージに appli_connector の port 番号を付加することにした。

ユーザは本システムを利用するにあたり、appli_connector の port 番号とどのア

図 2.5: port 番号を付加した UDP/IP パケット

プリケーションに対応するものであるかを定めておき、2.4 節で示す方式で通信を行う。これにより、図 2.3 の設計でもパケットの分類が可能となる。

なお、netpoll の詳細は 3 章で、netpoll_connector と appli_connector の詳細は 4 章で、netcontroller_io の詳細は 5 章で述べることにする。

2.3.3 target 上のソフトウェア

target で動かすカーネルは Linux-2.6.15 とした。

2.2 節で示したように、kgdb と netpoll が起動するバージョンである必要がある。netpoll はバージョン 2.6.5 で実装され、以降のバージョンはいずれでも使用することができる。

バージョン 2.6.5 以降で kgdb が対応するのは 2.6.13,2.6.15,2.6.16 のみである。このうち 2.6.16 に対応する kgdb は開発途中のバージョンであるため、Linux-2.6.15 を用いることにした。

target 上で動かすアプリケーションは kgdb と mingetty である。この 2 つのアプリケーションについて、以下詳説する。

kgdb

kgdb [7] とは、カーネルのデバッグを行うためのデバッガである。kgdb は、target の Linux カーネル内に kgdb を組み込んで host 上で動作する gdb から通常はシリアル端子により通信する。これにより、target 上の Linux カーネルを C 言語ソースレベルでリモートデバッグすることを可能としている。

host 上で動作する gdb は、target 上で動作する kgdb と通信することで、以下のデバッグ機能を実現する。

- C ソースレベルでのデバッグ

  カーネルとデバイスドライバを C ソースレベルでデバッグすることができる。また、関数呼び出しのトレースなども可能である。

- Thread のデバッグ

  target 上で動作している thread の情報として、thread の ID 番号、実行している関数名と引数、ソースファイル名と行番号などの情報を調べることが可能である。

  また、注目する thread を切り替えることで、その thread が実行している箇所のソース表示やレジスタ情報の参照も可能である。

- 例外のフック機能

  カーネル内で例外が発生した場合、kgdb に制御が移る。kgdb は例外をフックする設定をしておくことにより、カーネル内で例外が発生すると kgdb に制御を移し、gdb との通信が可能になる。これにより gdb 経由で例外発生原因の調査や究明ができるようになる。

- kernel asserts マクロ機能

  カーネル実行時に false になると kgdb へ制御を移す条件マクロ (KGDB_ASSERT) を記述することができる。コード中にこのマクロを記述することにより、false の検出や gdb 経由での原因調査が可能になる。

mingetty

mingetty [6] とは、login 認証を行う login やシェルの起動を任されているアプリケーションである。この機能があるため、開発者は host から target にログインし

たり、target 上でコマンドを実行することができる。

このような処理を行うアプリケーションは他にも getty や agetty など様々なものがある。このような中で mingetty を選んだのは、必要最小限の機能に絞られているため、ハードウェア資源消費量が少ないからである。target は組込み計算機というハードウェア資源の乏しい環境を想定しており、資源消費量は少ないことが望ましい。

2.3.4 host 上のソフトウェア

host のカーネルには、Linux-2.6.18 を用いた。カーネルデバッグを行うにあたり、host では kgdb への入出力に用いられる gdb が起動すれば足り、target の OS とバージョンを同一にする必要はないからである。ただし、host は target 上で動作するカーネルのソースを所持しなければならない。kgdb はソースレベルでのデバッグを行うことが可能だが、そのためにはカーネルとソースの対応付けが必要だからである。

なお、host 上で動作させるアプリケーションは netcat と gdb である。

netcat

netcat とは汎用 TCP/UDP 接続コマンドラインツールである。接続先 IP アドレス（もしくはホスト名）と port 番号を指定することにより、その接続先の該当する port 番号で待ち受けているサーバアプリケーションと通信することができる。またリッスンモードでは、指定した port 番号で通信を待ち受け、接続してきたクライアントアプリケーションと通信することも可能である。

今回は、mingetty との通信用に使用した。

gdb

gdbは現在広く利用されているデバッガであり、アプリケーションなどのデバッグを行うことができる。またkgdbと通信を行うことでカーネルデバッグを行うこともできる。

今回は、target上で動作するkgdbとの通信を行うために用いる。

2.4 通信の流れ

本システムはネットワークを用いてコンソールからの操作とカーネルデバッグを行うことを目標としているが、これを達成するためにはコンソールからの操作とカーネルデバッグを行うための通信経路が必要である。しかし、図2.3に示すように、netpoll_connectorとnetpoll間は1本の経路となっている。2.3節で示したように、netpollは複数起動することができないからである。このような設計をしたため、netpoll_connectorとnetpoll間でデバッグ用とコンソール用のパケットが混同してしまい、このままでは適切なアプリケーションにデータを渡すことができなくなるといった問題が発生する。

この問題を解決するため、パケット内のメッセージの先頭にappli_connectorのport番号を付加し、どのアプリケーションからパケットが届いたのか、識別することにした。

以下、送信(hostからtargetに対して通信)する場合と受信(targetからhostに対して通信)する場合について詳説する。

2.4.1 送信

図2.6に示すように、送信(hostからtargetに対して通信)する場合は

1. host上のアプリケーション

2. appli_connector
3. netpoll_connector
4. netpoll

の順にパケットが送信される。以下では、この流れに沿って詳説する。

host のアプリケーションから appli_connector へ

まず、ユーザは適当な port 番号を指定して appli_connector を起動する。その後、アプリケーションのパケット送信先をこの port 番号にして起動し、appli_connector と通信を行う。

appli_connector から netpoll_connector へ

appli_connector はアプリケーションからのパケットを受け取ると、そのパケットのメッセージの先頭に appli_connector の port 番号を追加した後、netpoll_connector に送信する。port 番号を追加したのは、target 上の netpoll が host 上のどのアプリケーションから送られてきたパケットなのかを識別するためである。netpoll への送信は netpoll_connector がまとめて行うため、パケットの port 番号を見ただけでは識別できない。

netpoll_connector から netpoll へ

netpoll_connector は appli_connector からのパケットを受けとると、そのまま target の netpoll に転送する。

netpoll

target 上の netpoll_connector からパケットが届いた場合、netpoll の受信処理関数が起動する。この受信処理関数は、パケット内のメッセージから appli_connector

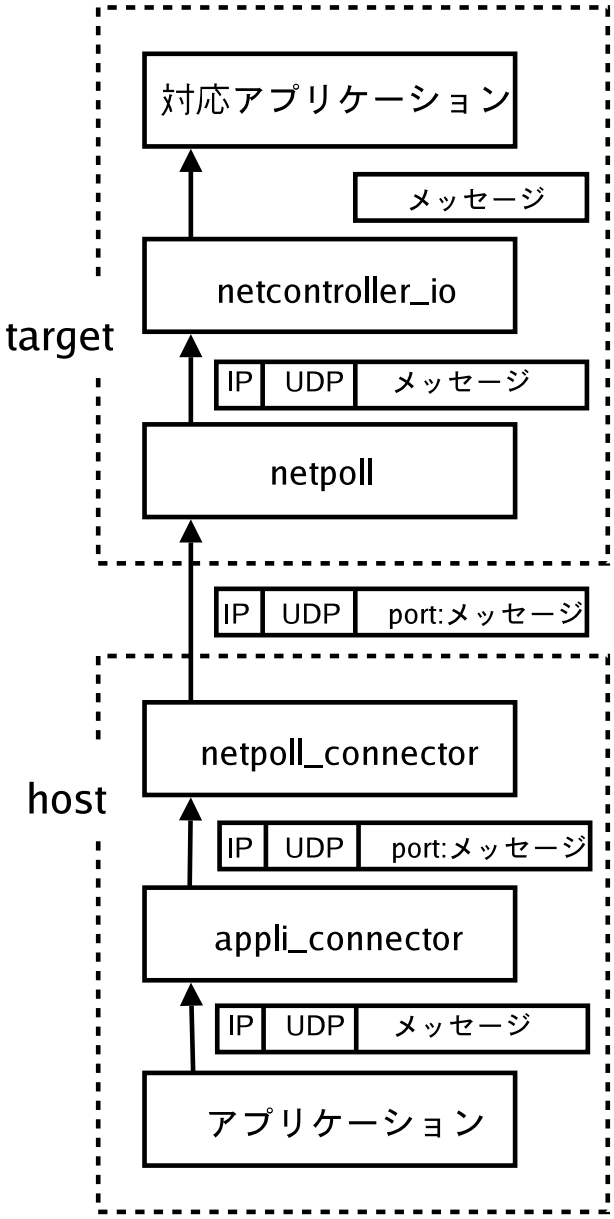

図 2.6: host から target に対して通信する場合

の port 番号を読み取る。これにより、host 上のどのアプリケーションからのパケットかを識別することが出来る。識別後は、各 host のアプリケーションに対応する target 上のアプリケーションのためのバッファに port 番号を取り除いたメッセージを追加する。netcontroller_io がこのバッファからメッセージを取得し、target の対応アプリケーションに受け渡す。

2.4.2 受信

図 2.7 に示すように、受信 (target から host に対して通信) する場合

1. target 上のアプリケーション
2. netpoll
3. host 上の netpoll_connector
4. appli_connector
5. host 上の対応アプリケーション

の順に通信される。以下、この流れに沿って説明する。

target 上のアプリケーションから netpoll へ

アプリケーションは netcontroller_io の write などを利用して、各アプリケーションに対応するバッファに文字を追加する。各アプリケーションに変更を加えることを避けるため、この際には対応する appli_connector の port 番号は追加されない。

netpoll から netpoll_connector へ

バッファが一杯になった場合、または、送信要求があった場合、netpoll の送信用関数 netpoll_send_udp を利用し、netpoll_connector への送信が行なわれる。送信の際、対応する appli_connector の port 番号がメッセージの先頭に追加される。

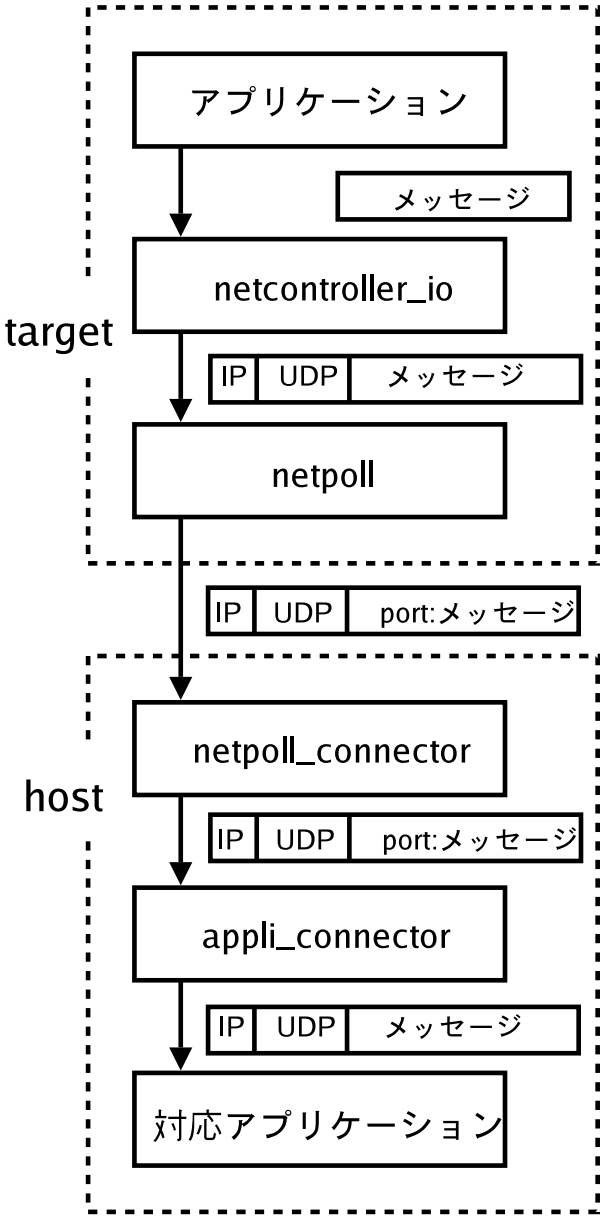

図 2.7: target から host に対して通信する場合

netpoll_connector がどの appli_connector に送信すればいいか、判断できるようにするためである。

netpoll_connector から appli_connector へ

netpoll から送られたパケットには、送信すべき appli_connector の port 番号が付いている。そこで netpoll_connector は、この port 番号を読み取り、適当な appli_connector に送信を行う。

appli_connector から host 上の対応アプリケーションへ

受信の場合の appli_connector の対応は、対応アプリケーションの port 番号が既知であるかにより分かれる。対応アプリケーションの port 番号は定まっていないことが多く、このような場合に送信しようとしても対応アプリケーションに到達することはなく、パケットロスとなってしまう。そこで、port 番号が不明な場合、port 番号が分かるまでバッファに溜めておくことにした。

対応アプリケーションの port 番号を設定出来る場合、または、対応アプリケーションからパケットが届き port 番号が既知となった場合は、パケットから appli_connector の port 番号を取り除いた後、対応アプリケーションに送信する。

第 3 章

netpoll

本章では、システム構成要素の 1 つである netpoll の概要や問題点、実装などについて述べる。なお、netpoll は既存のものであり、本システムはこれを用いて構成した。

3.1 netpoll とは

netpoll とは、Linux-2.6.5 になりネットワークスタックに追加されたものである。netpoll の目的は、完全なネットワークサブシステムと I/O サブシステムを利用できない状況で、カーネルがパケットを送受信できるようにすることである [8]。

ここでネットワークスタックとは、連携して動作する TCP,UDP/IP プロトコル層を実装したソフトウェアである。

本システムは host と target 間でパケット通信を行うため、target 上にもパケットの送受信を行う機構が必要である。この送受信には通常のネットワークスタックを用いることも可能である。しかし上記のメリットから、図 3.1 に示すように、本システムでは netpoll を用いて host の netpoll_connector と通信を行うことにした。

3.2 パケット送受信できる理由

netpoll は不安定な環境下でもカーネルがパケットを送受信することを目的としているが、これを実現するため通常とは異なる方法で UDP パケットの送受信を行っ

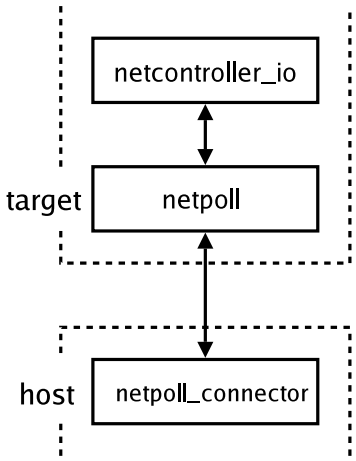

図 3.1: netpoll の配置

ている。

パケット送信については、netpoll を用いた場合も通常の場合も大体は同じである。送信はそれほど複雑な処理ではない。

一方で、受信は複雑な処理になりやすい。パケットがいつ届くのかは不明であり、パケット受信後も送信先 port への割り振りといった処理をしなければならないからである。

まずネットワークインタフェースで受信したパケットは、デバイスドライバのハードウェア割込み処理でデバイスドライバ内に取り込まれる。その後、このパケットをカーネルの受信キューに積み、ソフトウェア割込みを発生させる。

ソフトウェア割込みハンドラは、受信キューに積まれているパケットを取り出し、該当するプロトコルの受信ハンドラを呼び出す。該当するプロトコルの受信ハンドラは、プロトコルの仕様に応じて適切な処理を行い、アプリケーションにパケットが届けられる。

このような複雑な処理がされる中で、一部でも障害が発生したらパケット受信は行えなくなる。特に低レイヤの開発では通常のアプリケーション開発に比べカーネル内部に深く関わるため、その危険性は高くなる。

そこで netpoll では、デバイスドライバがカーネルの受信キューにパケットを積む前にパケット受信処理を行うことにした。具体的にはパケットがイーサネットのデバイスドライバ内に取り込まれると、このパケットが netpoll 宛てのパケットであるか送信元の IP や port 番号を基準に判断する。netpoll 宛てのパケットであれば netpoll の受信処理関数である rx_hook が処理を行う。これにより、複雑な受信処理機構に障害が発生したとしてもパケット受信を行うことができる。

3.3 問題点

本システムが考える、低レイヤでの組込み計算機のソフトウェア開発はカーネルに及ぼす影響が高く、ネットワークスタックに障害を生じる可能性もある。このような状況であっても、パケットの送受信を行うことができる netpoll は、十分利用する価値があると考えられる。

しかし、特殊な送受信を行う都合上、netpoll には以下の問題がある。

- あらかじめ設定した IP としか通信できない
- 待ち受ける port 番号は 1 つだけである
- 同一ホスト内で 1 つしか netpoll を動かすことが出来ない

これらは、netpoll の送受信が例外的な処理により実現されていることに起因する。例外を増やせば、通常のネットワークスタックを用いる通信のスループットが下がってしまう。そのため netpoll には拡張性がなく、また、本質的な問題点であるため今後のバージョンアップで改善されるとは考え難い。

そこで、本システムでは 4 章で示す appli_connector と netpoll_connector を用いて、netpoll の拡張性の問題を解決した。

3.4 netpollに関する実装

本節では、netpollを使用するために行った実装について説明する。

なおnetpollは動的モジュールとしてではなく、カーネル内に組込むことにした。カーネルデバッガは動的モジュールがロードされる前のカーネル起動時から使用されるが、その際にI/Oとなるnetpollが組込まれている必要があるからである。

そのため、カーネルコンパイルの設定を行う際にはnetpollをカーネル内に組込むように変更しなければならない。

3.4.1 初期化関数

netpollを使用できる状態にするための初期化関数では、まずhostやtargetのIP等の設定について確認を行い、それらが適切な場合はnetpollを有効化するためのnetpoll_setup関数を用いて初期化処理を行う。

netpollが有効化された後、次に本デバイスがキャラクタデバイスとして利用される、netcontroller_ioの登録作業をregister_chrdev関数を用いて行う。

最後に、kgdbのデバイス操作関数としても利用されるnetcontroller_ioの登録処理をkgdb_register_io_moduleを用いて行い、初期化処理を終える。

3.4.2 終了関数

終了関数では、初期化関数で有効化したnetpollなどを削除する処理を行う。

3.4.3 受信関数

3.2節に示したように、hostからtargetのnetpollが待ち受けるport番号にパケットが送られた場合、最終的には受信パケット処理関数であるrx_hookが実行される。rx_hookが実行される際には、引数としてパケット送信元であるnetpoll_connectorのport番号やパケット中のメッセージが渡される。

このメッセージの先頭にはappli_connectorのport番号が記載されているため、これに基づきhost上のどのアプリケーションがパケットを送信してきたのか判別を行う。そして、host上のアプリケーションに対応するtarget上のアプリケーション用の受信バッファにport番号を削除したメッセージを積む処理を行う。

なお、host上のgdbからの初回パケットは、target上でのkgdb起動を要求するパケットである。初回パケットのメッセージには特別な文字列が含まれるため識別可能である。

この初回パケットが届いた場合は、tasklet_schedule関数によりkgdbの実行がスケジュールされ、その後の実行によりkgdbが起動される。

3.4.4 送信関数

パケット送信にはnetpoll_send_udpという関数を使用する。引数として、送信するメッセージやその長さなどがある。詳しくは第5章で示すが、netpoll_send_udpはwrite関数など出力を行う関数により呼び出される。

第 4 章

connector

本章では、システム構成要素の 1 つである connector の概要や実装について述べる。なお、connector は本システム構築にあたり新規に作成したものであり、ユーザ空間で動作する。

4.1 connector の概要

connector は、図 4.1 に示すように、host 上のアプリケーションと target 上の netpoll 間のパケット中継をするものであり、appli_connector と netpoll_connector から構成される。

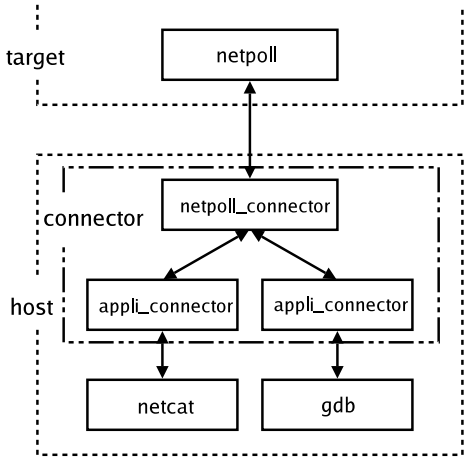

図 4.1: connector の配置

2.3 節で示したように、appli_connector と netpoll_connector は、

- 特定のportにしかパケット送信できないアプリケーションが、target 上のnetpollと通信するため
- 単一のportがnetpollと通信するようにするため

にそれぞれ導入したものである。

この2つのconnectorをtarget上で動作させることも考えられる。host上のアプリケーションはパケット出力を行うことができるため、target上のappli_connectorと通信し、target内部でnetpoll_connectorやnetpollと通信を行えばパケットの分類もできるからである。

しかし、以下の理由からhost上で起動させることにした。

理由1 netpollはhostからのパケットしか受け取らない。

理由2 不安定な環境でも動作する、というnetpollの優位性が失なわれる。

理由3 カーネルデバッグ中はconnectorの動作が停止する。

理由1について説明すると、デバイスドライバはnetpoll宛てのパケットであるか判断するため、パケットの発信元IPなどのチェックを行う。その際、発信元IPがnetpoll初期化時に設定したhostのIPでない場合はnetpoll宛てのパケットではないと判断し、通常のネットワークスタックを使った受信のルーチンに入る。そのため、同一ホストからパケットを送信しても、netpollは受け取らないのである。

同一ホストからのパケットもnetpollで受信するようにnetpollのソースを改良することも可能ではある。しかし、本システム以外のアプリケーションがnetpollを使用する場合に不具合が発生する可能性があり、改良することは避けるべきだと考えた。

理由2について説明すると、connectorは通常のネットワークスタックを使いパケット受信するアプリケーションとして実装した。このようなconnectorを介して

通信すると、第3章で示した、不安定な環境でも動作する、というnetpollの優位性が失われることになる。

理由3について説明すると、本システムはカーネルデバッグにも対応する。このカーネルデバッグを行っている間は、カーネルデバッガが他のアプリケーションの動作を停止させる。アプリケーションが動作すると、メモリ内容などが随時変更されるため、バグの原因特定が困難になるからである。そのため、カーネルデバッガ起動中はconnectorの動作も停止することになり、結果的にtargetでのパケットの送受信ができなくなると考えられる。

以上の理由から、これらのconnectorはhost上で動かすことにした。

以下、appli_connectorとnetpoll_connectorについて詳説する。

4.1.1 appli_connector概要

appli_connectorは、host上のアプリケーションとnetpoll_connectorのパケット中継を行うものである。appli_connectorは、host上のアプリケーションが固定のportとしか通信できない場合に、送信先port番号を変更できるようにするために導入した。

そのため、appli_connectorの受信port番号は可変なものとした。hostアプリケーションの送信先port番号に対応して、パケットの待ち受けを行えるようにするためである。なおhostアプリケーションの送信先IPが固定である場合は対応できないが、このような状況は考え難い。通常、IPはネットワーク環境等により変わるものだからである。

この他、appli_connectorで受信したパケット内のメッセージにappli_connectorのport番号を付加または削除する機能を実装した。2.3節で示したように、複数のアプリケーションと通信するためのパケット分類を行うのに、パケット内のメッセージ先頭にappli_connectorのport番号を付加し、これを用いてパケットの分類を行うことにした。そのため、パケット送信時におけるport番号の付加と受信時

における port 番号の削除を host 上でも行う必要があるが、これを appli_connector で行うことにした。

4.1.2 netpoll_connector 概要

netpoll_connector は appli_connector と target 上の netpoll のパケット中継を行うものである。

2.3.2 節に示したように、netpoll は送信先 IP を変更するのに初期化が必要という問題がある。また、パケット受信の際には、パケットの送信元 IP 等を見て netpoll 宛てのパケットであるか判断する。このパケット送信元 IP の判定は、netpoll の送信先 IP と同一 IP であるかによって行われる。

ところで、本システムは 1 対多の通信が行えるようにすることを考えている。これを行うにあたり、この IP の問題を解決する必要がある。netpoll は例外的な手続きによりパケットの送受信を可能としていることから、netpoll 自体の改良により複数の計算機との通信が可能になるとは考えられない。

そこで、netpoll との送受信を全て受け付ける netpoll_connector を作成した。netpoll_connector を使用することに伴い、appli_connector の送受信先は netpoll_connector とすることにした。

なお、netpoll_connector が中継することで、netpoll に届くパケットの発信元 port 番号は全て netpoll_connector のものになり、パケット発信元 port 番号による識別が出来なくなる。そこで appli_connector と netcontroller_io で、その appli_connector の port 番号または対応する appli_connector の port 番号をパケット内のメッセージに付加し、これを基に識別を行うことにした。

4.2 connectorの実装

本節では、appli_connector と netpoll_connector の実装について述べる。

4.2.1 appli_connector の実装

appli_connector は、host アプリケーションと netpoll_connector の中継を行うものである。appli_connector の処理の流れを図 4.2 に示す。

appli_connector にパケットを送信するのは、host 上のアプリケーションと netpoll_connector である。

host 上のアプリケーションからパケットが届いた場合、このパケットは本来 target 上の対応アプリケーションに宛てたパケットである。そのため本文中にあるのは対応アプリケーションに宛てたメッセージだけである。そこで、このメッセージ先頭に appli_connector の port 番号を付加した後、netpoll_connector にパケットを送信する。

netpoll_connector からパケットが届いた場合、このパケットは host 上のアプリケーションに届けられるものであるが、本文先頭に当該 appli_connector の port 番号が付加されている。netpoll_connector でパケットの分類を行うためのものであり、本来は不要のものである。この port 番号が付加されたまま host のアプリケーションにパケットを送信してもエラーとなる。そこで、この port 番号を削除した後に送信する。

4.2.2 netpoll_connector の実装

netpoll_connector は、netpoll と appli_connector の中継を行うものである。netpoll_connector の処理の流れを図 4.3 に示す。

netpoll_connector にパケットを送信するのは、appli_connector と target 上の netpoll である。appli_connector からパケットが届いた場合、port 番号の付加はすでに行われているため、netpoll へ送信するのみである。

netpoll からパケットが届いた場合、本文先頭には送信すべき appli_connector の port 番号が付加されている。そこで netpoll_connector は、この port 番号を基にパ

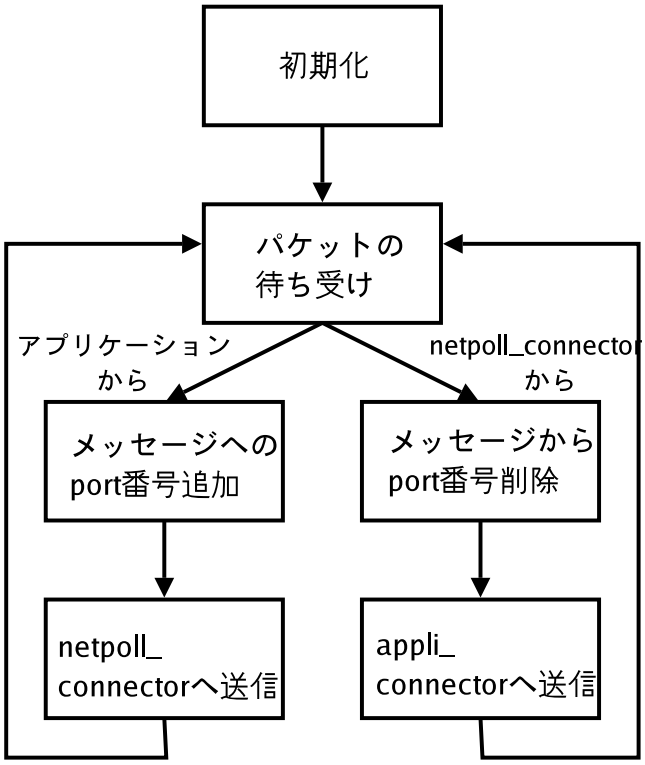

図 4.2: appli_connector の処理の流れ

ケット送信先を決定し、送信する。

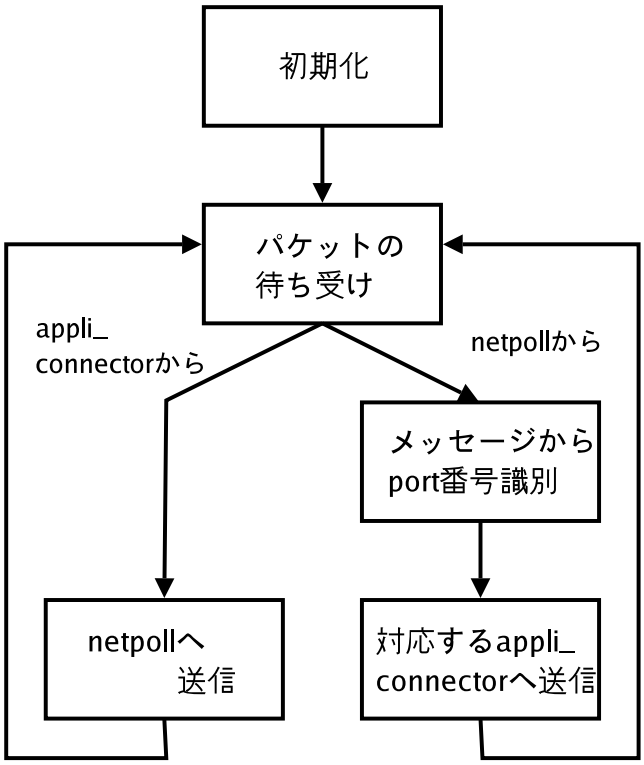

図 4.3: netpoll_connector の処理の流れ

第 5 章

netcontroller_io

本章では、システム構成要素の 1 つである netcontroller_io の概要や実装について述べる。なお、netcontroller_io は本システム構築にあたり新規に作成したものであり、ユーザ空間で動作する。

5.1 netcontroller_io とは

netcontroller_io とは、ファイルなどを操作する関数へのポインタを保存する構造体であり、一般にはデバイスドライバと呼ばれる。netcontroller_io は図 5.1 に示すように、mingetty や kgdb により利用され netpoll を用いて host との通信を行う。

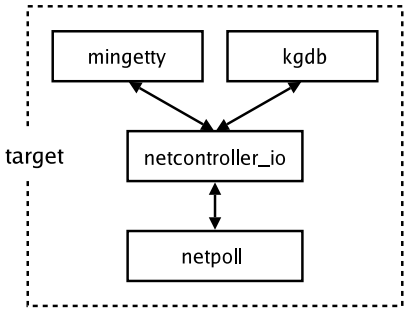

図 5.1: netcontroller_io の配置

Linux では、デバイス操作はデバイスドライバ内の各操作関数を経由し行われる。カーネルはすべてのデバイスを一環した方法で扱い、同じインタフェースを通してアクセスできる、等のメリットがあるからである。

target 上で動作する mingetty や kgdb はデバイスファイルを介して処理を行うため、このような構造体が必要である。mingetty や kgdb に対応する構造体として file_operations や kgdb_io がすでにあるため、当初はこれらを用いることを考えた。

しかし、複数の構造体を使おうとした場合、最後に有効化した構造体の関数が他方の構造体の関数への操作要求を奪ってしまう、という現象が発生した。具体的には file_operations、kgdb_io という順で有効化した場合に、file_operations の write 関数の実行要求があったとする。この場合に、kgdb_io の pre_exception が実行される場合があった。

このような不具合を回避するため、file_operations と kgdb_io を統合した netcontroller_io を作成し、netcontroller_io 経由でファイル操作等を行うようにした。

なお、コンソールやカーネルデバッグ以外の用途に利用する場合、この netcontroller_io の拡張を行えばよい。

5.2 netcontroller_io の実装

第 2 章で述べたように、本システムはコンソールからの操作とカーネルデバッグに対応することを目標としている。そこで、これらの処理に利用される file_operations と kgdb_io の関数の実装を行った。

なお、netcontroller_io はモジュールではなく、カーネル内に組込むものとする。kgdb はカーネル起動時に初期化処理をするが、その際に処理関数への構造体が必要だからである。そのため、kgdb は netcontroller_io を用いてデバッグを行うようにカーネルコンパイルされるための設定を drivers/net/Kconfig に追加した。

以下、実装した関数について詳説する。

5.2.1 file_operations

コンソールで必要な操作を行う関数群である。以下、実装した関数について説明する。

write

write 関数はデバイスファイルへメッセージを書き込むための関数である。

mingetty は host 上の netcat にメッセージを書き込む際に write システムコールを発行する。write 関数は引数として、書き込むデータが格納されているバッファと書き込むメッセージ数をとる。

netcat に対応する appli_connector の port 番号を書き込むメッセージの先頭に追加し、netpoll の送信用関数である netpoll_send_udp を用いて、netpoll_connector にパケットを送信する。

read

read 関数はデバイスからメッセージを読み込むための関数である。mingetty は host 上の netcat からデータを読み込む際に read システムコールを発行する。read 関数は引数として、読み込んだデータを格納するバッファと読み込むメッセージ数をとる。

netpoll の受信処理関数である rx_hook によりコンソールの受信用バッファにメッセージが積まれるまで待ち、カーネル関数 copy_to_user を用いて mingetty にメッセージを渡すという流れになる。

open

open 関数は、ドライバが後の操作に備えて初期化を行うための関数である。

本システムは通信に UDP を用いるが、UDP はコネクションレスの通信を行うためコネクションを張るなどの初期化は不要である。そのため、実質的処理は行って

いない。つまり、target との接続は必ず成功した事になるが、本研究で考えるネットワーク環境は十分な帯域があり、パケットロスは発生しない環境を考えており、問題は無いと考える。

release

release 関数は、ドライバが操作を終え、事後処理を行うための関数である。

本システムは通信に UDP を用いるが、UDP はコネクションレスの通信を行うためコネクションを閉じるなどの事後処理は不要である。そのため、実質的処理は行っていない。

ioctl

ioctl 関数は、デバイスのパラメータを操作するための関数である。

本デバイスはキャラクタデバイスとしても利用されるが、その際の出力レートや表示画面の大きさなどを設定する。mingetty はこれらの値を基に出力を行うため、表示画面が崩れることがなくなる。

5.2.2 kgdb_io

kgdb がカーネルデバッグを行う上で必要な操作に関する関数群である。以下、実装した関数について説明する。

read_char

read_char 関数はデバイスから 1 文字メッセージを読み込むための関数である。

具体的には、netpoll の rx_hook によりカーネルデバッガの受信用バッファにメッセージが積まれるまで待ち、その後このバッファから 1 文字メッセージを取り出し、kgdb に渡すという流れになる。

write_char

write_char 関数はデバイスに 1 文字のメッセージを書き込むための関数である。メッセージの host への送信は kgdb_flush を用いて行う。

具体的には、カーネルデバッガの送信用バッファに 1 文字分のメッセージを積む処理を行う。このバッファが一杯になる場合、kgdb_flush を呼び出し メッセージを送信する処理を行う。

init

init 関数は、kgdbが netcontroller_io を使うための初期化を行うための関数である。

具体的には、kgdb の I/O 登録用関数である kgdb_register_io_module を用いて netcontroller_io の登録を行う。

kgdb_flush

kgdb_flush 関数は、カーネルデバッガの送信用バッファに積まれたメッセージを送信するための関数である。kgdb_flush は kgdb により明示的に、または送信用バッファが一杯になった場合に write_char により呼び出される。

具体的には、書き込むメッセージの先頭に appli_connector の port 番号を付加し、netpoll の送信用関数 netpoll_send_udp を用いて target に向けてメッセージの送信を行う。

pre_exception

pre_exception 関数は、カーネルデバッグを行うにあたり前処理をするためのものである。

具体的には、ARP 要求への返信や意図しない発信先からのパケットの廃棄を行うためのフラグを立てる。

post_exception

post_exception 関数は、カーネルデバッグを行った後処理をするためのものである。

具体的には、pre_exception 関数で立てたフラグを下げる。

第 6 章

動作確認

本章では、第 2 章から第 5 章で設計実装したシステムの動作確認を行った。

6.1 動作確認環境

本節では、本システムを動作確認環境について述べる。本システムは図 6.1 のような環境で動作確認した。host と target は表 6.1 のものを用いた。target は組込み計算機を想定しているが、今回の動作確認では組込み計算機の代用としてノート PC を用いた。

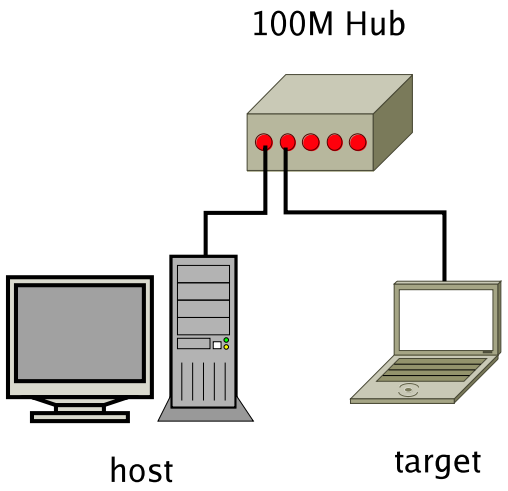

図 6.1: 動作確認に用いた計算機環境

表 6.1: 動作確認に用いた計算機

	host	target
OS	Linux 2.6.18	Linux 2.6.15
CPU	Intel Pentium 4 3.4GHz	Mobile Intel Pentium 4 1.8GHz
メモリ	4096 MByte	512 MByte
ネットワーク	10/100/1000 Ethernet	10/100 Ethernet

なお、host 上で動作させたアプリケーションは netcat 0.7.1 と gdb 6.4、また target 上で動作させたのは mingetty 1.07-1 と kgdb 2.4 である。

6.2 使い方

本節では、本システムやコンソール、カーネルデバッガの使い方について説明する。なお、host と target の IP、各アプリケーションの port 番号は表 6.2,6.3 のように設定した。

まず準備として host 上でデバッガ用とコンソール用の appli_connector と netpoll_connector を起動する必要がある。表 6.2 の対応する port 番号を引数に、それぞれのアプリケーションを起動させる。

```
./appli_connector 6442 && ./appli_connector 6443 && ./netpoll_connector 6441
```

なお、target 内の netpoll と netcontroller_io は、カーネル内に組込むようにしているため、カーネル起動と同時に使用可能であり、別途起動する必要はない。

表 6.2: host 側の設定

IP	host	192.168.166.31
port	netcat	6542
	gdb	起動のたびに変動
	netcat 用 appli_connector	6442
	gdb 用 appli_connector	6443
	netpoll_connector	6441

表 6.3: target 側の設定

IP	target	192.168.166.85
port	netpoll	6440

6.2.1 コンソールの使い方

host 側では、接続先を host 上の netcat に対応する appli_connector とし、また待ち受ける port 番号等の設定をした netcat を起動し、target との通信を行えるようにする。

```
netcat -u -l -p 6542 192.168.166.31 6442
```

target 側では、作成したデバイスに対応するデバイスファイルを引数に mingetty を起動する。

```
mingetty /dev/netreceive
```

6.2.2 カーネルデバッグの行い方

host にて target 上で動作しているカーネルを指定して gdb を立ち上げる。

```
gdb linux-2.6.15.5-netreceive/vmlinux
```

その後、通信先として host の gdb に対応する appli_connection を指定し、target 上の kgdb を起動させるコマンドを入力する。

```
(gdb) target remote udp:192.168.166.31:6443
```

この操作により、target 上での kgdb 起動を要求する旨のパケットが target に送信され、kgdb が起動しカーネルデバッグが行える状態になる。

6.3 簡単な評価

本節では、コンソールとカーネルデバッガを 1 つずつ動かした場合に意図した動作を行うか、という観点から評価を行った。

具体的には、それぞれの主要コマンドを入力し、正しく実行されるかで確認を行った。

6.3.1 コンソール

コンソールで行う主要操作である、ファイル操作とテキスト編集を一般の計算機で利用する仮想端末と同様に行えるか、という観点から評価した。

ここでファイル操作とは、ファイルの読み取りや書き込みなどを行う操作であり、ls などにより実行される。またテキスト編集とは、テキストファイルに文字の記述や削除を行うことで、vi や emacs により行われる。

ログイン後、ls、cd、pwd した結果を図 6.2 に示す。図 6.2 より、それぞれのコマンドを実行した場合に期待される結果が表示の崩れを起こすことなく出力されていることが分かる。表示速度もシリアル端子経由の場合と比べ、遜色無いものであった。

他のコマンドについても試したが、入力した文字を target と通信するには Enter が必要であること、Ctrl-c を入力すると netcat 自体が終了してしまう、等により操作に不都合が生じる場合があった。これらの不都合はいずれも netcat 自体の仕様によるものであり、netcat の改良により改善可能である。

続いて、vi、emacs の実行を行った。表示が崩れたり、表示速度が遅いということもなく、先の操作の不都合以外には問題は無かった。

以上より、本コンソールは正常に動作していると言える。

6.3.2 カーネルデバッガ

本節では、シリアル端子経由でカーネルデバッグを行っていたのと同様にイーサネット経由でもカーネルデバッグが行えるか、という観点から動作確認を行った。

利用する状況としては 2.3.3 節でも示したが、target 上の全 thread から 1 つを選び、その thread について対応する C のソースを表示し、スタックを表示することで thread の起動経緯を調べることにする。

target 上で動作する全 thread の一覧については、

```
hama@target:~$ ls
Desktop                                 mingetty-1.07
kernel                                  mingetty-vi-log
kgdb-patch                              mnt
kgdblog                                 nc
less                                    netreceive
linux-2.6.15.5                          netreceiver
linux-2.6.15.5-netreceive               nfsboot
linux-2.6.15.5-netreceive.tar.gz        portchanger
linux-2.6.15.5.tar.gz                   sarge
linuxdrive3                             vmlinux
hama@target:~$ cd linux-2.6.15.5-netreceive
hama@target:~/linux-2.6.15.5-netreceive$ pwd
/home/yoshi/linux-2.6.15.5-netreceive
hama@target:~/linux-2.6.15.5-netreceive$ ls
COPYING        Makefile        arch     fs       lib      security
CREDITS        Makefile~       block    include  mm       sftp.cmd
Documentation  Module.symvers  bzImage  init     net      sftp.cmd~
Kbuild         README          crypto   ipc      patches  sound
MAINTAINERS    REPORTING-BUGS  drivers  kernel   scripts  usr
```

図 6.2: ファイル操作中の表示

(gdb) info threads

で取得することができる。

(gdb) thread n

により、調査する thread を選ぶ。

(gdb) list

で選択した threadがスケジューリングされている場所を中心に 10 行のソースが表示される。引数を与えることで、他の箇所のソースを調べることも可能である。

(gdb) backtrace

により、この thread に関するスタックの追跡結果を最深部から順に表示する。

それぞれの実行結果を図 6.3、6.4、6.5 に示す。

図 6.3、6.4、6.5 の結果は、シリアル端子経由の場合と同等であったため、正常にカーネルデバッグが行えていると考えられる。

6.4 並行動作の評価

本システムは通信を行うのにパケットを用い、また、パケットのメッセージ先頭に appli_connector の port 番号を付加したことにより、複数のアプリケーション間で同時に通信できるようになった。そこで本節では、コンソールとカーネルデバッガと通信した場合に両者が意図した通りの並行動作を行うか評価を行った。

6.4.1 意図する動作

評価を行う状況は、target 上で mingetty を起動させた後、host の gdb から kgdb 起動要求のパケットを送信して target 上で kgdb を起動させる。そして mingetty と kgdb に操作コマンドを送信し、その動作を確認する。

```
(gdb) info threads
  86 Thread 2257 (uim-candwin-gtk)  0xc02e343a in schedule ()
    at kernel/sched.c:1684
  85 Thread 2255 (firefox-bin)  0xc02e343a in schedule ()
    at kernel/sched.c:1684
  84 Thread 2253 (firefox-bin)  0xc02e343a in schedule ()
    at kernel/sched.c:1684
         (中略)
  3 Thread 2 (migration/0)  0xc02e343a in schedule ()
   at kernel/sched.c:1684
  2 Thread 1 (init)  0xc02e343a in schedule ()
   at kernel/sched.c:1684
* 1 Thread 32768 (Shadow task 0 for pid 0)  breakpoint ()
   at kernel/kgdb.c:1888
(gdb) thread 85
[Switching to thread 85 (Thread 2255)]#0  0xc02e343a in schedule ()
    at kernel/sched.c:1684
1684            switch_to(prev, next, prev);
```

図 6.3: thread 情報の取得

```
(gdb) list
1679                    WARN_ON(rq->prev_mm);
1680                    rq->prev_mm = oldmm;
1681            }
1682
1683            /* Here we just switch the register state and the
 stack. */
1684            switch_to(prev, next, prev);
1685
1686            return prev;
1687    }
1688
```

図 6.4: ソースの表示

```
(gdb) backtrace
#0  0xc02e343a in schedule () at kernel/sched.c:1684
#1  0xc02e3c3e in schedule_timeout (timeout=Variable "timeout" is
  not available.) at kernel/timer.c:1093
#2  0xc0137fe1 in futex_wait (uaddr=Variable "uaddr" is not
  available.) at kernel/futex.c:699
#3  0xc0138247 in do_futex (uaddr=146390632, op=0, val=Variable
 "val" is not available.
) at kernel/futex.c:832
#4  0xc013832d in sys_futex (uaddr=0x8b9be68, op=0, val=11,
  utime=0xb2a95348,uaddr2=0xb, val3=-1297525876) at
  kernel/futex.c:876
#5  0xc02e46fd in syscall_call () at include/linux/jiffies.h:338
#6  0xc033ded5 in end_of_stack_stop_unwind_function ()
#7  0xc033ded5 in end_of_stack_stop_unwind_function ()
```

図 6.5: thread のトレース結果

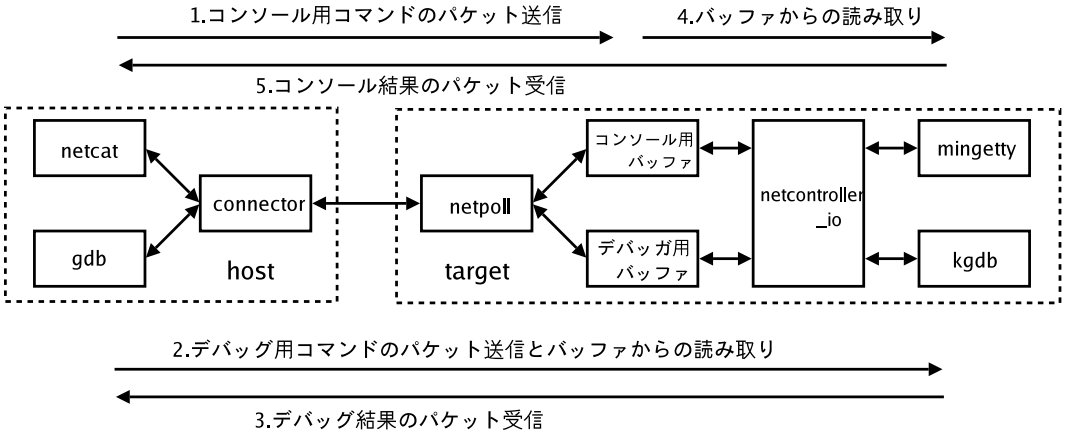

図 6.6: 意図する動作

このような場合に予想される動作を図 6.6 に示し、以下にその説明を行う。

1. host の netcat から送信された mingetty 用のコマンドは、target の netpoll により受信され、コンソール用バッファに積まれるが、target の mingetty により実行はされない。taregt で起動中の kgdb が他のアプリケーションの動作を停止させるからである。アプリケーションが処理を進めることでメモリ等の状態に変化が生じ、正確なカーネルデバッグが行えなくなることを防ぐためである。

2. host から送信された kgdb 用コマンドは、target の netpoll により受信され、デバッガ用バッファ、netcontroller_io を通じて kgdb に渡され、実行される。kgdb 自身の動作は制限されていないからである。

3. target の kgdb は、実行した操作の結果を host の gdb に送信する。2,3 のような動作は kgdb が終了するまで続く。

4. kgdb が終了すると、他のアプリケーションの動作制限が解除されるため、mingetty はコンソール用バッファから実行するコマンドを取り出し、実行する。

5. mingettyは、実行した操作の結果をnetcatに送信する。

6.4.2 実際の動作

本節では、前節の予想通りに動作するか確認を行った。

mingettyを起動させtargetへのログインを行い、コマンド待ち状態(netcontroller_ioのread関数実行中)になったところで、kgdbを起動させた後、6.4.1節に示した動作を行った。その結果を図6.7に示し、以下に説明する。

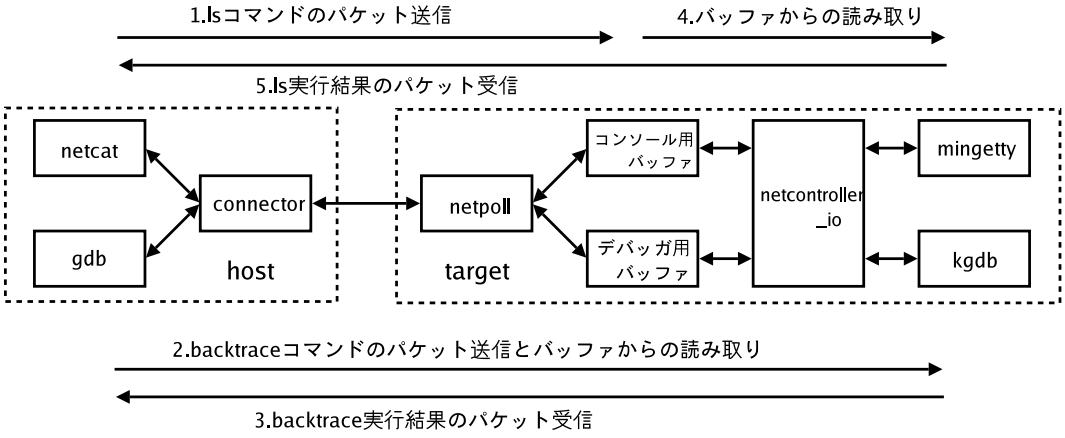

図 6.7: 実際の動作

1. hostのnetcatからtargetのmingettyに向けてlsコマンドを送信した。targetのnetpollにより受信され、コンソール用バッファに正しく積まれたことを確認したが実行はされなかった。

2. 続いてhostのgdbからbacktraceコマンドをtargetに送信した。netpollにより受信され、kgdbに渡され実行されたことを確認した。

3. その後、kgdbにより実行されたbacktraceの結果がgdb上で表示された。表示された結果は、図6.5と同等であったため、正しく実行されたと考えられる。

4. gdb から exit コマンドを送信し kgdb を終了させると、その後 mingetty がコンソール用バッファから ls コマンドを取り出し、実行した。

5. その後 netcat で ls コマンド実行結果を表示した。なお、この実行結果は図 6.6 の実行結果と同一であった。

以上より、意図する動作と実際の動作が一致したため、正しく並行動作がされたと言える。

6.5 ネットワーク周りのデバッグ評価

本システムでは host と target 間の通信にイーサネットを用いた。そのため、本システムを用いてネットワーク周りのデバッグが行えるか、という疑問が生じる。本節では、ネットワーク周りの処理がなされた場合にカーネルデバッガを起動できるか、という観点から評価を行った。

ここでネットワーク周りとは、target 上のアプリケーションからパケット送信要求がありイーサネットデバイスからパケット送信がなされるまで、host からのパケットをイーサネットデバイスが受信し対応するアプリケーションにパケットが渡されるまでの間の処理を意味する。

6.5.1 動作確認方法

ネットワーク周りのデバッグが可能かを評価するため、6.5.1 節に示すパケット受信に関する関数に breakpoint を設置し、これらの関数が呼び出された場合にカーネルデバッガが起動するかを確認した。カーネルデバッガが起動すればデバッグは行えるからである。

受信関数に breakpoint を設置したのは、以下で説明するように、netpoll と通常のネットワークスタックを用いた受信処理の流れが明確であり、カーネルデバッグ可能範囲を特定するのが容易となるからである。

なお breakpoint の設置は

(gdb) b breakpoint を設置する関数名

で行うことができる。breakpoint 設置後は、一旦デバッグ作業を抜ける。その後、breakpoint を設置した関数が呼び出されればカーネルデバッガが起動しデバッグを行うことができる。

パケット受信の流れ

以下では、まず、パケットを受信した際の処理の流れについて説明する。

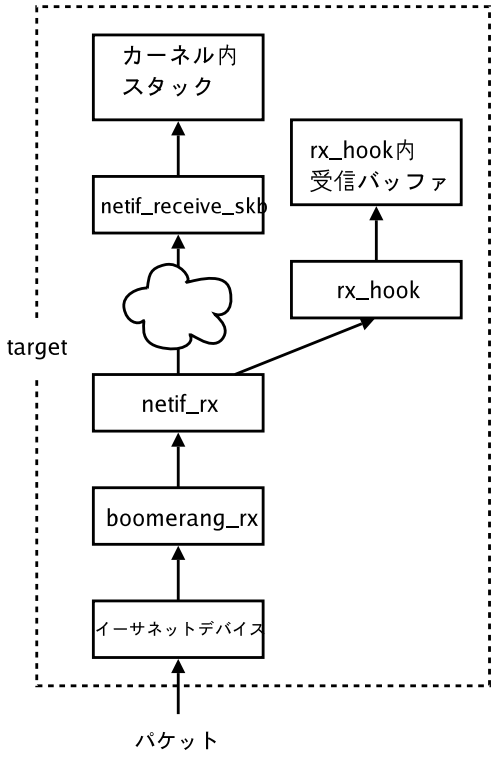

図 6.8: パケット受信時の処理の流れ

図 6.8 に示すように、target のイーサネットデバイスがパケットを受信すると、そのデバイスドライバのハードウェア割込みを発生し、そのパケットをデバイスドライバ内に取得する。この処理を行うのが boomerang_rx 関数である。

パケット取得すると、netif_rx が netpoll 宛てのパケットであるか判別を行う。netpoll 宛てのパケットである場合、受信パケット処理関数である rx_hook が呼び出される。rx_hook は、3.4.3 節に示すように、受信したパケットを送信したアプリケーションを特定し、それに対応する受信バッファにメッセージを積む関数である。

netpoll 宛てのパケットでない場合、このパケットを受信キューに積み、受信ソフトウェア割込みを発生させ、キューからパケットを取り出し、パケットタイプに対応する上位レイヤに渡す、などの通常の処理がいくつかの関数により実行されることになる。このうち、上位レイヤにパケットを渡す処理を行うのが netif_receive_skb 関数である。

6.5.2 動作確認結果

本節では、6.5.1 節で示した関数について、breakpoint を設置し、これらの関数が起動されるよう host からパケットの送信を行った際の結果を示す。

netif_receive_skb

図 6.9 に示すようにデバッグは正しく行うことができた。すなわち、netif_receive_skb の呼び出しによりデバッガが起動し、この関数に対して 6.3.2 節のような処理を行うことができた。

rx_hook

図 6.10 に示すように、カーネルデバッグを行うことはできなかった。

詳細であるが、シリアル端子経由のメッセージから netpoll 宛てのパケットであるか判別を行い rx_hook を呼び出す netpoll_rx は実行されたことを確認したが、rx_hook 実行の際のメッセージは表示されなかった。また、その後はシリアル端子経由でもイーサネット経由でも何の通信も行われず、デバッグを行うことはできな

```
(gdb) target remote udp:192.168.166.31:6443
Remote debugging using udp:192.168.166.31:6443
breakpoint () at kernel/kgdb.c:1888
1888            atomic_set(&kgdb_setting_breakpoint, 0);
(gdb) b netif_receive_skb
Breakpoint 1 at 0xc028f248: file net/core/dev.c, line 1587.
(gdb) c
Continuing.

[New thread 1574]
[Switching to thread 1574]
Breakpoint 1, netif_receive_skb (skb=0xdf166e80) at
net/core/dev.c:1587
1587            if (skb->dev->poll && netpoll_rx(skb))
(gdb)
```

図 6.9: netif_receive_skb へ breakpoint を設置した場合の結果

```
(gdb) target remote udp:192.168.166.31:6443
Remote debugging using udp:192.168.166.31:6443
breakpoint () at kernel/kgdb.c:1888
1888            atomic_set(&kgdb_setting_breakpoint, 0);
(gdb) b rx_hook
Breakpoint 1 at 0xc029ff15: file drivers/net/netreceive.c, line 66.
(gdb) c
Continuing.


 (以後、無反応)
```

図 6.10: rx_hook へ breakpoint を設置した場合の結果

かった。

boomerang_rx

boomerang_rx に breakpoint を設置した場合も、rx_hook と同様の現象が発生し、カーネルデバッグを行うことができなかった。

考察

rx_hook や boomerang_rx は kgdb が通信を行う際に利用される関数である。そのような関数に breakpoint を設置したことで、kgdb の通信に不都合が生じ、カーネルデバッガが起動しなかったと考えられる。

一方、netif_receive_skb は kgdb により利用されることはない関数である。そのため、kgdb の通信には影響がなく、kgdb はデバッグを行うことができたと考えられる。

以上のことから、本システムを用いたカーネルデバッグはイーサネットデバイスドライバなど、netpoll より下のレイヤでの開発には用いることができないという問題がある。しかし、イーサネットデバイスドライバより上のレイヤにおける開発、例えば通常のネットワークスタックを利用したデバイスの開発には利用することができる。

Linux ではイーサネットドライバは多数存在し、組込み計算機の開発ではそれらを利用するため、実際の利用場面において問題はないと言える。

第 7 章

関連研究

本章では関連研究や本研究と関連のあるアプリケーションについて述べる。

7.1 netconsole

netconsole [9] は netpoll を用いて UDP でメッセージを送信する。host 側からの入力には対応せず厳密にはコンソールとは言えないが、起動ログなどの出力に用いられている。

host 側からの入力には対応しておらず、本研究で考えるコンソールとして利用することはできない。また、シリアル端子の置き換えを行うものでもない。

7.2 Etherconsole

Etherconsole [10] はシリアル端子経由のコンソール入出力をイーサネットインタフェースへリダイレクトする。このような研究やアプリケーションの開発は多々行われている [11]。

このシステムはサーバでの利用を前提としており、コンソールにのみ対応しているため、カーネルデバッグなどコンソール以外の用途に用いることができない。

また論文中での記述は無いが、このような実装では複数のアプリケーション間で同時に通信することはできないと考えられる。どのアプリケーションからのパケットなのか判別することができないからである。

以上より、このシステムでは本研究で考える目的を達成することはできないと考えられる。

7.3 remoteconsole

ネットワークを利用する組込み計算機向け開発資源拡張システム [12] の一環としてコンソールとして利用できる remoteconsole などが実装されている。このシステムは組込み計算機というハードウェア資源の乏しい環境を考慮し、ハードウェア要求が低い、という特徴を有する。

ネットワーク経由で他の計算機資源を利用することを目的をしており、本研究の目的であるシリアル端子の置き換えを達成することはできないと考えられる。

7.4 Parrot

Parrot [13] とは、分散コンピューティングでリモートのプロセッサやリモートのデータにアクセスするためのライブラリである。通信の中継を行う、という点で本システムと類似するところがある。

Parrot を利用するには、このライブラリを呼び出すようにアプリケーションを書き換える必要があるが、本システムではこのような必要はない。

7.5 kgdboe

kgdboe [7] は netpoll を使い、イーサネット経由で kgdb を用いたカーネルデバッグを行うためのアプリケーションである。

カーネルデバッグ以外の用途に利用することができず、また複数のアプリケーション間の通信を可能とするような機能もないため、本研究の目的を達成することはできないと考えられる。

上記のアプリケーションを複数動かすことも考えられる。しかし 3.3 節で示したように、netpoll は複数起動に対応しておらず、このような手法での解決を図ることはできない。この問題について、通常のネットワークスタックを用いて複数通信することも考えられるが、3.1 節や 6.5 節で示したメリットが得られない点で、本研究には意義があると考える。

第 8 章

今後の課題

本章では、今回設計実装したシステムについて、今後改良すべき課題について述べる。

8.1 TCP への対応

target 上でパケット通信を行う netpoll が UDP のみ対応するため、host 上の appli_connector や netpoll_connector、アプリケーションは全て UDP での送受信を行うこととした。この host 上のアプリケーションには、TCP での通信のみに対応する実装の場合がある。

ここで、netpoll 自体が今後のバージョンアップにより TCP に対応する可能性もある。しかし、パケットが通常のネットワークスタックを用いて処理される前に例外的に処理されていることから、コネクションの確立など複雑な処理を要する TCP に対応するとは考え難い。

そこで、appli_connector か netpoll_connector に TCP パケットと UDP パケットの違いを吸収する機構を設ける。具体的には、コネクションの確立や切断のためのパケットは target に送信せず、データ通信のためのパケットは target に UDP で送受信し、TCP でアプリケーションに送信する。

8.2 1 対多通信への対応

本システムでは、イーサネットによるパケット通信を行うこととした。netpoll と通信できるのは netpoll_connector に限られるため、図 8.1 のようなシステム構成をとることが考えられる。このような場合に、どの host からのパケットかを判断するためパケットの先頭に IP を付加する方法がある。このような通信方式をとればパケットの送受信は可能である。

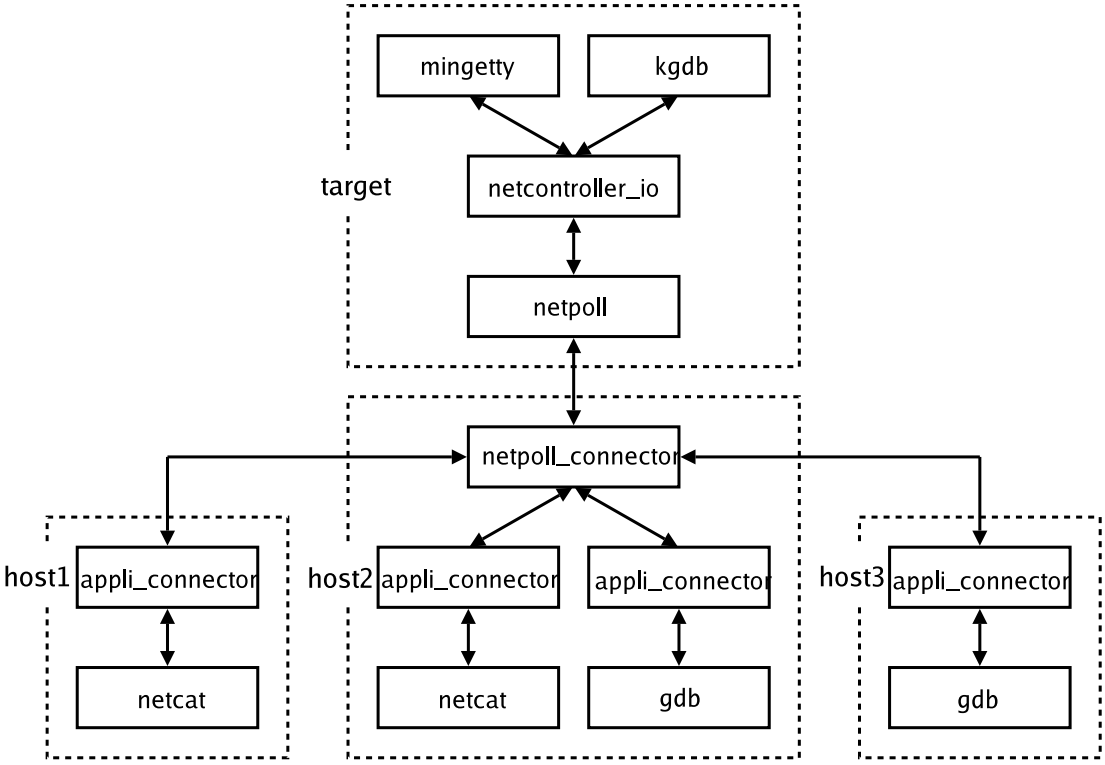

図 8.1: 1 対多通信するためのシステム構成

1 対多の通信を行う場合、複数の host が 1 つのアプリケーションに通信するケースと各 host ごとにアプリケーションを起動しそれと通信するケースがある。

前者については、実行結果をこのアプリケーションと通信している host に送信する必要がある。その際、送信する先の host を覚えておく、または前もって設定しておくための実装変更が必要である。

後者については、現在の実装を延長させた場合は、アプリケーション用のバッファの数等が増え、プログラムの保守などに苦労が生じることが予想される。そのため、target 側の実装に変更が必要である。

第 9 章

まとめ

本研究は、既存のシリアル端子を経由して行われていた通信をイーサネットを経由した通信に置き換えるために必要なシステムを設計した。そして、コンソールとカーネルデバッグにおける通信の置き換えを行うための実装を行った。

具体的には、

- コンソール用アプリケーション、カーネルデバッガの処理で呼出されるデバイスドライバ netconnector_io
- netpoll を使用するための機構
- host 上のアプリケーションと target 間のパケット中継を行う appli_connector と netpoll_connector

を設計実装し、パケット発信元のアプリケーションを特定するための通信方法を定めた。

また、作成したシステムについて

- コンソールとデバッガを単独で動かした場合に正常動作を行うか
- コンソールとデバッガを並列に動かした場合に正常動作を行うか
- ネットワーク周りのデバッグを行うことができるか

という観点から動作確認実験を行った。この結果、単独、並列共に正常動作を行うこと、デバッガのパケット送受信に用いられないネットワークの処理についてはデバッグ可能であることを確認した。

本システムは、シリアル端子を有さない組込み計算機に対して、イーサネットを用いてコンソールからの操作やカーネルデバッグを行うこと可能とした。

組込み計算機のソフトウェア開発は低レイヤでも行われるが、コンソールやカーネルデバッグは、特に低レイヤでの開発に利用されるものである。そのため、本システムを用いることで、製品版のようなシリアル端子を有さない組込み計算機を対象としたソフトウェア開発を行うことができる。

近年は、開発用計算機として利用される通常の計算機にもシリアル端子が付属しないものが多いが、このような計算機を利用した開発にも対応することができる。

謝辞

本研究を遂行するにあたっては、いろいろな方々にお世話になりました。

まず、指導教員の多田好克先生には日頃から熱心なご指導、そしてご鞭撻を賜わりました。ご多忙中にもかかわらず論文の草稿を丁寧に読んで下さり、大変貴重なご助言をいただきました。ここに厚く御礼申し上げます。また、厳しくも貴重なご意見、適切なご助言を頂いた佐藤喬助手に感謝いたします。

そして、本研究が行なえたことは、研究方針や方法論について議論をし、共に研究生活をおくってきた多田研と村山研の学生諸氏おかげでもあります。最後に、これらの皆さんに感謝いたします。

参考文献

[1] 組込み Linux 入門,TECHI,Vol.16,CQ 出版社,pp.6–68.

[2] SoftwareDesign,2005 年,12 月号, 組込み Linux 実践講座,pp.16–63.

[3] 梅本昌典, 乃村能成, 横山和俊, 谷口秀夫, 丸山勝巳:“走行モード変更機構を利用したデバイスドライバの実現”, 情報処理学会研究報告, 2006-OS-101, pp.25–31(2006).

[4] 谷口秀夫, 乃村能成, 田端利宏, 安達俊光, 野村裕佑, 梅本昌典, 仁科匡人: “適応性と堅牢性をあわせ持つ Ant オペレーティングシステム”, 情報処理学会研究報告, 2006-OS-103, pp.71–78(2006).

[5] Joel R. Williams:“Embedding Linux in a Commercial Product: A look at embedded systems and what it takes to build one”, Linux Journal, Volume.1999, Issue 66es(1999).

[6] mingetty http://sourceforge.net/projects/mingetty/

[7] kgdb,kgdboe http://kgdb.linsyssoft.com/

[8] Alessandro Rubini, Jonathan Corbet 著/ 山崎 康宏, 山崎 邦子, 長原 宏治, 長原 陽子 訳,“Linux デバイスドライバ 第 3 版”, オライリージャパン (2005).

[9] netconsole

http://www.linux.or.jp/JF/JFdocs/kernel-docs-2.6/networking/netconsole.txt.html

[10] Kister,M., Hensbergen,E.V. and Rawson, F.: “Console over Ethernet”, Proceedingsof the FREENIX,Track:2003,USENIX Annual Technical Conference, pp.125–136(2003).

[11] Fairisle network console

http://www.cl.cam.ac.uk/Research/SRG/bluebook/18/netcon/netcon.html

[12] 明神智之, 佐藤喬, 多田好克: “ネットワークを利用した組込み計算機向け開発用資源拡張システム”,(2006).

[13] Parrot:Transparent User-Level Middleware for Data-Intensive Computing