

異種OSのシミュレート方式の一手法

安田 剛

株式会社 東芝 システム・ソフトウェア技術研究所

本報告では、組み込みソフトの開発時に目的のプログラムをホストマシンでデバッグ・テストを行うために必要となるターゲットOSのシミュレーション法について検討する。ここでは、高級言語で書かれた組み込み用プログラムのうちターゲットOSのシステムコールを使用している部分をホストマシン上でも動作させるようなモジュールをライブラリーとして提供する。そしてその構成法について論じる。本方式は、インプリメントの容易さと、テスト・デバッグのやりやすさを目的としている。

A Simulation Method of System Calls in Another OS

Takeshi YASUDA

Systems & Software Engineering Laboratory, Toshiba Corporation
70 Yanagi-cho, Saiwai-ku, Kawasaki 210, Japan

yasuda@ssel.toshiba.co.jp

This paper provides how to simulate system calls which a target operating system(OS) has on a cross environment. An application program using target OS's system calls can execute on a cross environment by using "OS simulate library" proposed in this paper. And how to build its library is described. This proposition will enable developing application programs using system calls on the cross environment.

1. はじめに

ソフトウェアの開発環境は、CASE (Computer Aided Software Engineering) ツールに代表されるようにそれ自身の開発、改良もさかんに行われている。このような、開発工程の各フェーズを支援するツールを取り揃え、統合化した環境は、開発者の作業効率を高めるのに役立つ。

一方、マイクロエレクトロニクスの急速な発展によりマイクロコンピュータ (マイコン) を組み込んだ製品が市場に溢れ出してきている。このようなマイコン組み込み製品の開発では、ソフトウェアとハードウェアの開発が並行して行われることが多い。組み込み用のソフトウェア開発の環境は、ホストコンピュータ上でクロスソフトウェアを用いる環境すなわちクロス環境で行われることになる。クロス環境では先のCASEツールを用いられるので都合がよい。

しかし、クロス環境におけるプログラム開発で問題の1つであるデバッグについて注目してみると、組み込みソフトのための決定版となる手法が確立していない。そのため、開発者はデバッグに悩みながらも独自のノウハウを集め、作業を進めているという現状がある。

また、開発用の言語についてみると、CPU (Central Processing Unit) の高性能化と開発期間の短縮化によって、高級言語指向の過程を踏んでいる。なかでもC言語は、組み込みシステム用高級言語としても要求が高くなっている。そのため、アプリケーション・ソフトを開発するものにとっては、C言語さえ知っていれば、ターゲット用のプログラムであるかないかということさはさほど気にならない場合が多くなっている。

さらに、組み込みシステムのソフトウェア構成は、従来の単一構成からカーネル (オペレーティング・システム) 層、アプリケーション層といった階層構造をとる傾向にある。これによって、システム内の資源管理に関してアプリケーション・プログラム開発者は解放されることになり、開発効率や保守の容易性の向上が期待できる。

ここで、組み込み用のアプリケーション・プログラ

ムにおいて、オペレーティング・システム (Operating System: OS) の機能を、そのインターフェースを用いて使うものが増えつつある。システムコールと呼ばれるOSの機能を使うためのモジュール群を使用するものである。このようなプログラムをクロス環境上でデバッグすることを考えてみる。システムコールの実体はターゲット環境上で動作するように作成してあるために、そのままクロス環境上で動かすことはできない。クロス環境上でもターゲット用OSのシステムコールの動作を何等かの方法で、シミュレートすることができれば、デバッグ効率が大幅に向上することが期待できる。

そこで、本報告では、ターゲット用OSのシステムコールを用いた高級言語のプログラムを、クロス環境上でデバッグするための方法として、システムコールの部分をクロス環境上で疑似的に実行するOSシミュレート方式について提案し、その構成法について述べることにする。

以下では、まずクロス環境上でのソフトウェア開発について、使用されるツールを中心にセルフ環境と比較する。次に、クロス環境上でのシステムコール・シミュレーションについて考察し、本方式の概要を述べる。そして、本方式でのシミュレーションを行うモジュール群の構成法について述べる。

2. クロス環境でのソフトウェア開発

組み込みシステム用のソフトウェアは、ホストマシン上のクロスソフトウェアを用いて開発される場合が多い。その流れは、図1に示すように、セルフ環境のそれとほとんど同じである。使われるツールは、エディタ、コンパイラあるいはアセンブラ、ライブラリアン、リンカ、デバッガが中心となる。

一般にマイコン開発でのデバッグ手法には、以下のよう大別して3通りの方法がある。

① デバッグ・モニタ

デバッグ専用のモニタプログラムを用いてターゲットシステムとホストマシンとの通信によりデバッグを進めていく方法。

② ICE (In-Circuite Emulator)

専用のハードウェアを使用し、ターゲットシステムのCPU部分をエミュレートしながらデバッグを進めていく。実機デバッグに使用する。

③クロス・デバッグ（ソフトウェア・シミュレータ）

ホストマシン上でターゲットシステムの環境をシミュレートしつつ、目的のプログラムのデバッグを行う。

①、②の手法は、ターゲットシステム（実機）を使用するためにシステムテストの段階では必ず必要になってくる方法であるが、その準備としての時間的・経済的な手間が掛かる。③はそのような手間が掛からず容易に論理的なテスト、デバッグを行うことができる。本報告が扱う手法は③に関連するものである。

これまで組み込みソフトウェア用のシミュレータやデバッグは、主にバイナリレベル（機械語、アセンブラのレベル）のものが多く、これらは、ターゲットシステムのハードウェアをシミュレートするものであり、現在様々なものが存在し、関連技術も研究されている(1)、(2)。

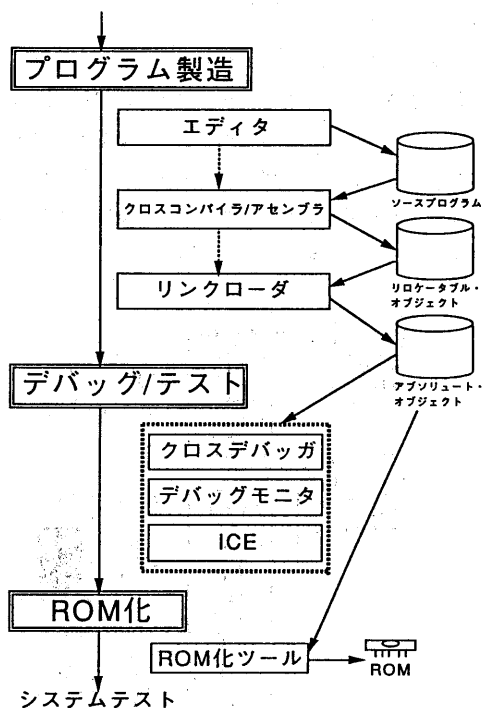


図1. 組み込み用ソフトウェア開発の流れ

バイナリレベルのデバッグにおける問題点は、CPUごとにデバッグを作成する必要があることや、OSが管理するリソースの状態をユーザに表示することができないためにOSレベルでのデバッグ・テストに關する効率が低下することにある。

我々の提案は、それらバイナリレベルのシミュレータの構築法とは若干異なるアプローチをとる。1章でも述べたように高級言語を使って組み込み用アプリケーションプログラムを開発する人にとっては、資源へのアクセスもシステムコール経由で行うことになる。そこで、バイナリレベルではなくシステムコールのインターフェースレベルでシミュレーションを行い、論理的なテストやデバッグを進めようとするアプローチをとる。

OSのシステムコール・インターフェースに関しては、文献[4]～[9]等の研究が行われている。これらは、UNIXなどターゲットOSのシステムコールあるいはOSそのものを異なるOS上で完全に動作させようとするものであり、既存ソフトウェアの継承利用を目的としている重要な研究である。本報告での提案方式は、できるだけ簡単なインプリメント方法と、組み込み用アプリケーション・ソフトウェアの論理テスト・デバッグをよりおこないやすくすることを重視しているために目的を異にする。

3. クロス環境上でのシステムコール・シミュレーション

3.1 クロス環境上でのシステムコールのデバッグと理想的環境

ここでは、クロス環境上での組み込みソフトウェアのデバッグについて考える。

いま、図2のようなプログラムをクロス環境上でデバッグする事を考える。但し、このプログラムの言語仕様は、ある規格（例えば、C言語の場合のANSI規格など）が存在し、クロスコンパイラ、セルフコンパイラともその規格を満足しているものとする。

ここでは①の行でターゲットOSのシステムコールを使っている。この場合はSystemXがシステムコールの名前であり、Arg1が引数である。

このプログラムをホストマシンのセルフコンパイラでコンパイルし、ロードモジュールまで作る。

この際、①の行では、ターゲットOSのシステムコールを使っているが、このプログラムをクロス環境上で動作させるためには、このシステムコールと同じ動作をするようなモジュールをホストマシン用にも用意し、このプログラムへリンクさせなければならない。

システムコールを異なる環境上でシミュレートさせるモジュールを作成するには、以下のような理由により実現が困難となる。

- i) システムコールのソースコードが入手不可能、あるいは入手困難である。
- ii) システムコールはハード資源と密接な関係にあり、クロス環境とターゲット環境はハード構成が異なっていることが一般的である。
- iii) ターゲットOSとセルフOSの機能の過不足のため、インターフェースのみの置換には限界がある。

従来こういった問題に対しては、クロス環境上にターゲットOS自身をインプリメントし、その2次的なOS上で目的のプログラムをデバッグするという方法がある(図3)。この方法は、理想的であり、完璧に近いデバッグが行えるが、実現を考えると新たにOSを開発するのと同等の手間がかかるし、ホスト環境の能力がターゲットOSの能力より著しく低い場合は実現不可能であるといった欠点がある。

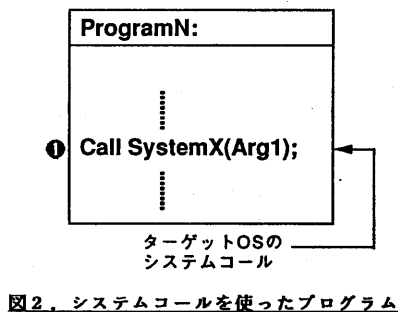


図3. 理想的な方法

3. 2 システムコールのシミュレート

3. 1 で述べたことを克服し、解決するためにここで提案する方式のねらいは以下のようなことである。

- ①実現のし易さ
- ②既存のツールをなるべく有効に使う
- ③クロス環境を選ばない

①の実現性は、いうまでもないことであるが、この部分に開発時間が取られてしまえば本末転倒である。②は、クロス環境上の有効なツールをなるべく利用することにより開発と習熟の効率化を狙う。③は、クロス環境のOSなどが違っていても同じようなシミュレーションが行えるようにすることである。

次に、システムコールのシミュレート方式について提案する。なお、対象とするプログラムはすべて高級言語とし、OSのシステムコールはその言語とのインターフェースを介して使用できる。また、クロス環境にあるセルフコンパイラの言語仕様とクロスコンパイラの言語仕様は、互換性があるものとする。

本方式では、通常クロス環境上ではできないシステムコールの実行をクロス環境上で疑似的に動かそうとするものである。

図4は、本方式でデバッグを行うまでの手順である。目的とするプログラムをクロス環境上のプログラムとしてコンパイルし、そのプログラムのなかで使用されているターゲットOS用のシステムコール部分をシミュレートライブラリとリンクしクロス環境上で動作可能にしてデバッグを行う。

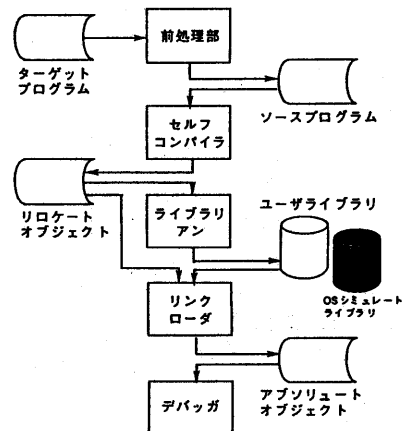


図4. 本方式でのデバッグ手順

4. シミュレート・ライブラリの構成法

4.1 本方式の基本的な考え方

本方式の主目的は、クロス環境における異種OSシステムコールのシミュレート環境を容易に構築することである。さらに、その環境をはかのクロス環境上でも移植が簡単に行えることを目指している。そのため本方式では、高級言語で作成される組み込み用アプリケーションソフトウェアで使用されるシステムコール部分のシミュレートモジュールの集合（ライブラリ）として提供する。

このライブラリ構成のための基本的なモデルを図5に示す。このモデルでは、シミュレータ、資源状態管理表、そしてデバッガからなる。それぞれ3つのオブジェクトが必要なメッセージをやり取りしながらシミュレーションをおこなっていく。シミュレータオブジェクトは、アプリケーション・プログラムから要求のあったシステムコールのシミュレーションを行うが、システムコールの操作対象である資源（論理資源、物理資源）自身の振る舞いをシミュレートするリソースシミュレータとそれ等の資源を管理するシステムコールシミュレータから構成される。資源状態管理表オブジェクトは、リソースシミュレータとシステムコールシミュレータの結果を保存しておく。デバッガオブジェクトは、ユーザとのインターフェースを受け持つ部分でありデバッグ用コマンドの実行及び実行結果の表示等とそれらのコマンドの機能動作を司る。そして、ほかのオブジェクトに対して必要なメッセージを送信する。

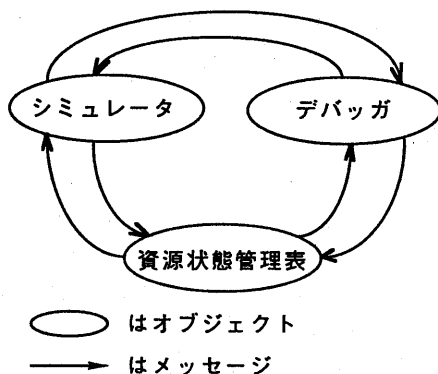


図5. 本方式の基本モデル

4.2 構成

ここでは、オペレーティング・システムのシステムコールをシミュレートするモジュールの集まりであるシミュレート・ライブラリの構成について述べる。

図6は本方式でのライブラリ構成図である。アプリケーションプログラムからOSインターフェース（システムコール）を通して、このライブラリ・モジュールとのやりとりが行われる。4.1で説明した基本モデルに従って、システムコール・シミュレータ、リソース・シミュレータ、資源状態管理表群、デバッガに分かれている。さらに、アプリケーション・プログラムと本モジュールとのインターフェース部分であるインターフェース解析部を設けている。以下で、各部についての説明を行う。

(1) インターフェース解析部

ここでは、アプリケーション・プログラムから渡されるパラメータなどを解析し、どのシステムコールのシミュレーションを行うかを調べ、それに必要なパラメータの解析とチェックを行う。また、今までに何回同じシステムコールが呼ばれたかをカウントしたりといった、デバッガが必要となるデータを取得し、デバッガあるいは、システムコール・シミュレータに対して起動のメッセージを送る。

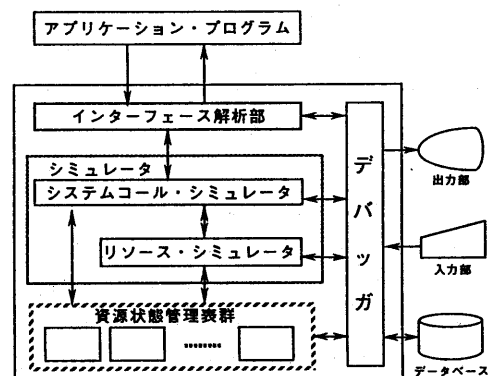


図6. シミュレート・ライブラリの構成

(2) システムコール・シミュレータ

ここでは、インターフェース解析部で解析されたインターフェース情報によって、該当するシステムコールのシミュレーションを行う。目的のシミュレーションを行うオブジェクト群で構成されるが、必要に応じて資源管理表群の該当するオブジェクトにデータをメッセージとして送ったり、逆にそこに書いてあるデータを管理表から送信してもらったりしながらシミュレーションを実行する。また、資源に関するシステムコールのシミュレーションは、リソース・シミュレータとのメッセージパッシングで、実現される場合もある。デバッガからは、実行停止・再開、状態データ送信等の要求メッセージが送られてくるので、その要求に応じた実行を行う。

(3) 資源状態管理表群

ここでは、OS内で使用される管理表のすべてがオブジェクトとして存在している。例えば、メモリ状態、タスク状態、I/Oデバイスの使用状態、ハードウェア・コンポーネントの内部状態などである。これらの表は、システムコール・シミュレータ、リソース・シミュレータ、デバッガからメッセージによってアクセスされる。ユーザからは、デバッガのヒューマン・インターフェースを通してアクセスされることになる。

(4) リソース・シミュレータ

ここでは、システムコール・シミュレータからのメッセージを受け取り、その内容に応じたOS資源の振る舞いをシミュレートする。シミュレーションを行った後の情報は、資源状態管理表群の該当する管理表オブジェクトにメッセージとして送られる。

ハードウェア構成要素の各コンポーネントはオブジェクトとしてモデル化されるため、このオブジェクトへの問い合わせによってシミュレートが行える。また、実時間性の問題に対してはシミュレートする時間をあらかじめデータベースへ登録しておきブレイク・ポイント間の実行時間を算出するか、ユーザとの会話によって設定できるようにする。なお、このオブジェクト部分が実現されない場合でもユーザとの会話により疑似的にシミュレートが行える。

(5) デバッガ

これは、本ライブラリ・モジュール内での、実行を制御する部分であり、ユーザとのヒューマン・インターフェースをも含んでいる。

本方式の特徴として、システムコールシミュレートモジュール内にデバッガ部を組み込んだことが挙げられる。各システムコールを呼び出したときにデバッガが起動するので特にシステムコールに関したデバッグ情報のみで良い場合は、専用デバッガが無くても良い。

ユーザからの、コマンドにより、ブレイクポイントまでの実行、各資源の状態の表示、操作シーケンスのロギングなどが行える。

4.3 設計指針

(1) インターフェース解析部

アプリケーションプログラムからのシステムコールに対して、最初にここへ到達するようにする。各システムコール名に対して必要なパラメータ数、パラメータ値をチェックし保存すると共に、実行制御部へメッセージを送り、システムコールをシミュレートして良いかどうかの判断を待つ。実行制御部からシミュレート許可ができれば目的のシステムコールシミュレートモジュールを呼び出す。

(2) システムコール・シミュレータ

システムコールシミュレータは、各シミュレートモジュール別に内部構造が異なってくる。ここでの問題は、

①ホストOSへの変換をどのようにするか。

②ホストOSに必要な機能が備わっていない場合の処理

③並行処理のシミュレート

等があげられる。①についてはあらかじめ調べておけばシミュレートモジュール内で変換すれば良い。②については、不足機能をインプリメントする必要がある。呼び出されるシステムコールの外部仕様などから実際のインプリメントを行う。③については、ターゲットOSとクロス環境のOS間のプロセス、あるいはタスクの概念を良く踏まえた上でインプリメントを行う必

要がある。

システムコール・シミュレータの設計では、目的OSのシステムコールを調べ、リソースの分類を行う。このとき、システムコールの操作対象をリソースとする。リソースの抽出が終わったら、クロス環境上のOSと比較し、同じものがあるかどうかを調べる。同じものがある場合は、クロス環境上のOSシステムコールで代用できないかを考える。代用できる場合は、それを用いることにする。

(3) リソース・シミュレータ

前節で本システムの全体構成を説明したが、ここではリソース・シミュレータの内部構成について検討する。

リソース・シミュレータは、論理資源(タスク、セマフォ、メールボックスなど)や物理資源(割り込みコントローラ、タイマ・コントローラなど)のシミュレータを基本的にソフトウェアで実現するところである。そのため、実現に際して、次の2つの問題が考えられる。

①対象資源の機能を完全にシミュレートできるか
(完全性)

②シミュレート時の時間的遅延

①については、対象資源の内部構成について精通していなければならない、その振る舞いを完全にシミュレートしようとするとは実現は複雑困難になる。このことは、特に物理資源をシミュレートする際に問題となるであろう。

②も①に関連するが、物理資源は高速動作が一般的であり、ソフトウェアでシミュレートする場合、その実行にかなりの時間がかかる。

本方式では、①、②の問題により目的の物理資源のシミュレートができない場合、以下のいずれかあるいはその組み合わせによってこの部分を構成する。

(I) シミュレートする物理資源がホストマシンの環境にあり、利用可能な場合はそれを使うことにする。文字どおりハードウェアでシミュレートするのである。例えば、クロス環境上のタイマ・コントローラで使われていないポートを使いシミュレートを行うことや、

拡張ボードに存在するメモリの一部をシミュレート用として使うことなどが挙げられる。この方法をとれば、上の問題は解決できる。

しかし、クロス環境にこういった目的で使用できる物理資源はあまり存在しないのが普通である。また、この方法では、他の環境上に移植する際、もう一度作り直す必要がでてくる。

(II) リソース・シミュレータのリターン情報を予めデータベースなどに格納しておき、入力パラメータの条件により適当な出力情報を返すようにシステムコール・シミュレータを作成する。ハード資源の内部動作をシミュレートせず、参照インターフェース部分の処理のみを行うのである。こうすることで、該当するリソース・シミュレータは必要なくなる(図7)。この方法を用いると、①、②の問題は、ある程度回避できると思われる。しかし、この処理部分の時間が、実際よりも長くなることも予想される為、完全には対処できないこともあるだろう。

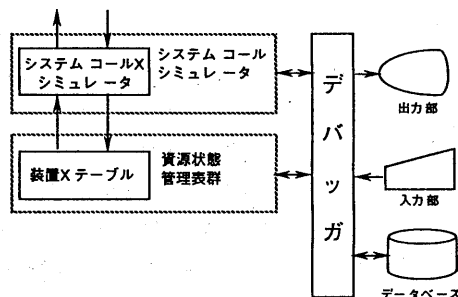


図7. リソース・シミュレータ無しの実現

(III) (II)での時間的遅延の問題の最終的な対策としては、本システム内での実行時間を計測する時計を持たせ、シミュレートにかかった時間を実物がかかった時間に替えることである。例えば、現時点からあるブレイクポイントまでの時間がシミュレートでは、30msecかかる場合、先の時計を実物の場合の時間10μsecに修正するのである。この方法で、時間的問題は解決できると思われる。この場合、アプリケーション・プログラムからインターフェース解析部に制御が移ってきた時刻とその逆の時刻つまり外部の時計を合わせることを考えなければならない。

5. おわりに

本報告では、クロス環境上において組み込み用のプログラムがターゲット上のOSのシステムコールを使用しているとき、そのシミュレーションを行う方法について説明した。本方式では、対象となる組み込み用のプログラムが、クロス環境となっているホストマシンのセルフコンパイラで処理されるソースプログラムに変換される。そのプログラムが参照しているシステムコールの部分はクロス環境上でシミュレートされるようにモジュール化され、リンクされる際にアブソリュート・モジュール内に組み込まれる。このアブソリュート・オブジェクトはホストマシンのセルフ環境上で実際に動作する。そのため、デバッグを行う場合には、セルフ環境上のデバッグが使用できる。このことは、専用のデバッグを作成する必要がなくなり、新たに命令などを覚えなくてもよく使用者の負担が軽減される。

また、シミュレート・モジュール内の構成を検討した。資源の状態を管理するテーブル群を中心にリソース・シミュレータ、システムコール・シミュレータ、インターフェース解析部、デバッグの各部にわけ、明確化した。ハードウェア・シミュレータの実現が困難な場合についての構成法についても検討を行った。

本方式では、システム・コール部分について完全なシミュレーションを行うのではなく、疑似的に対応することができるため、開発期間の短縮化ができ、比較的いろいろな環境上で実現ができるので、開発者の負担を軽減できる。

今後、この方式による処理系を実現し、評価する。さらに、より高度なシミュレーションが行えるようシステムコール・シミュレート部の汎用性を高める方式について研究を進めていく。

【参考文献】

- [1] 国峰幸雄: 汎用クロス・ソフトウェアのすべて, 電波新聞社, 1988. 4.
- [2] 井上勝博, 三原幸博: ハードウェア動作記述からのシミュレータ自動生成, 情報処理学会研究報告, Vol. 89, No. 81, 89-SE-68, 1989. 9.
- [3] 津田順司ほか: マイクロコンピュータ用モジュール構造リアルタイムOSの一構成法, 電子通信学会論文誌(D), Vol. J69-D, No. 2, pp. 156-169, 1986. 2.
- [4] 遠城秀和: 異種OS上におけるUNIXインターフェースの実現法, 情報処理学会第33回(昭和61年後期)全国大会講演論文集(I), 2V-2, pp. 271-272, 1986.
- [5] 松本匡通, 金田洋二: OSインターフェース整合のための仮想化実現方式の一検討, 情報処理学会第39回(平成元年後期)全国大会講演論文集(II), 6M-6, pp. 1028-1029, 1989.
- [6] 石橋奈津子ほか: CTRON仕様OSへのUNIXシステム搭載の一方法, 情報処理学会第39回(平成元年後期)全国大会講演論文集(II), 5P-3, pp. 1229-1230, 1989.
- [7] 竹内宏典, 工藤明彦: CTRON/UNIXインターフェースにおけるプロセス制御, 情報処理学会第39回(平成元年後期)全国大会講演論文集(II), 5P-4, pp. 1231-1232, 1989.
- [8] 工藤明彦, 竹内宏典: CTRON/UNIXインターフェースにおけるsignalの実現, 情報処理学会第39回(平成元年後期)全国大会講演論文集(II), 5P-5, pp. 1233-1234, 1989.
- [9] 遠藤秀和: マルチインターフェースOSにおけるクロスプロセス間通信, 情報処理学会第39回(平成元年後期)全国大会講演論文集(II), 6P-4, pp. 1251-1252, 1989.
- [10] K. H. Britton *et al.*: A procedure for designing abstract interfaces for device interface modules, in Proc. Fifth Int. Conf. Software Eng., IEEE Compt. Soc., Mar. 1978.
- [11] 高橋薫ほか: オブジェクトの概念に基づいたネットワークOSの設計法, 電子情報通信学会論文誌(D), Vol. J71-D, No. 10, pp. 2128-2139, 1988. 10.
- [12] 小菊一三ほか: 実時間指向ポータブルOSの構想, 電子通信学会技術研究報告, Vol. 86 No. 37, CPSY86-3, pp. 13-18, 1986. 5.
- [13] 安田剛, 三原幸博: クロス・デバッグ方式, 情報処理学会第40回(平成2年前期)全国大会講演論文集(II), 1R-1, pp. 1001-1002, 1990. 3.