クロス開発環境におけるホスト/ターゲット処理一体記述の提案とデバッグ環境への適用

若林隆行[†] 高田広章[†]

組込み機器の開発では，プログラムの開発環境（以下，ホスト）と実行環境（以下，ターゲット）が異なる場合がある．このような環境のテストプログラムは，ターゲット上の情報を用いてホスト上で実行される場合がある．このようなプログラムはデバッガを用いた操作を行う必要があるため本質的でない記述が増大し，また作成者にはデバッガに関する知識が必要となる．この問題に対し我々は，前述のようなホスト上で実行されるプログラムの一部と，ターゲット上で実行されるプログラムを一体に記述することを提案する．この記述の特徴は，一体記述されるホスト上のプログラムが，ターゲット上で実行可能な形式で記述される点にある．本提案により，開発者のプログラム記述量を抑えるとともに，記述を容易にする．本稿では，我々の提案の概要を示す．また具体的に提案をデバッグ環境へ適用し，記述結果の評価を行う．

Proposal to Integrate Host and Target Program for Cross-Development Environments and Applying it to Debugging Environments

Takayuki Wakabayashi[†] and Hiroaki Takada[†]

Most of embedded systems are developped with two separated environment, Development Environment (Host) and Execution Environment (Target). In such a case, there is a possibility that a debug program runs under a host computer in spite of handling data which a target computer contains. Knowledge of the debugger is necessary for developers to develop such a debug program because developers have to write a debug program with debugger's instructions due to handling data which a target computer contains. But, these tasks are inessential and enlarging a load of developers. To solve this problem, we propose describing host and target programs together regardless of difference of environment. The trait of this proposal is that a form of the host program described together is similar to a form of a target program. By using of our proposal, description of such a debugger program get easier and smaller. In this paper, we describe details of our proposal and the result of applying it to a debug process.

1. 背　　景

近年，組込みシステムの大規模化・複雑化にともない，開発期間や開発コストの増大に加え設計品質や信頼性の低下が大きな問題となっている．このような組込みシステムの開発には，クロス開発環境が用いられる[1)]．クロス開発環境とは，プログラム開発を行うコンピュータ（以下，ホスト）と，プログラムを実行するコンピュータ（以下，ターゲット）が異なるような環境を指す．組込みシステムは組み込む対象の機器に専用化して設計されるため，システムごとにハードウェア構成や周辺デバイスが異なる．しかしクロス開発環境はシステムのハードウェア構成に強く依存するため，ハードウェア構成が変わると使用できない場合がある．そのため，複雑化する機器に対してデバッガの対応が遅れていると指摘されている．

この問題に対し，我々はクロス開発環境におけるデバッグのための新しいソフトウェアアーキテクチャを提唱し，ITRON デバッギングインタフェース仕様を策定した．

図 1 に，ITRON デバッギングインタフェース仕様に準拠したデバッグ環境の構成図を示す．六角形はインタフェースを，矢印は機器間通信経路を示す．図内で用いている RIF および TIF という略称は，それぞれ RTOS アクセスインタフェースとターゲットアク

† 豊橋技術科学大学情報工学系
Department of Information and Computer Sciences, Toyohashi University of Technology

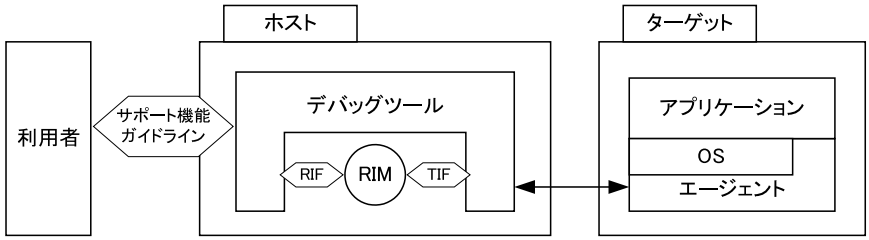

図 1 ITRON-DBIF 仕様に準拠したデバッグ環境
Fig. 1 Overview of the debugging environment which complied with ITRON-DBIF.

セスインタフェースを示す．それぞれの詳細に関しては，文献 2) および 3) を参照されたい．

ITRON デバッギングインタフェース仕様の特徴は大きく分けて 2 つある．

(1) 内部構造に依存しないデバッガ作成が可能

通常，デバッガの持つ機能のうち OS に依存する部分（以下，OS サポート機能）は OS 内部の変数などを操作するため，デバッガは OS の内部構造に強く依存する．そのため，外部仕様が同一でも内部仕様が異なるような複数の OS を一度にサポートすることは困難であった．この問題に対し，ITRON デバッギングインタフェース仕様ではデバッガの OS サポート機能を RTOS Interface Module（以下，RIM）と呼ぶモジュールに分離し，RIM を RTOS 作成者に提供させることで，デバッガ本体が OS の内部構造に依存することを回避した．デバッガは RIM を取り込むことで，OS の内部構造に依存せずに OS サポート機能を提供できる．ターゲットによる影響を最小限に抑えるため，RIM はホスト上に配置されている．

(2) 高い適応性を確保

本稿では，利用者が状況に応じてデバッグ環境の特性を変化させることができるという性質を適応性と呼ぶ．組込み機器は用途に応じて多様な制約があり，デバッグ時に求められる性能も用途によって変化する．この要求に対し，ITRON デバッギングインタフェース仕様では，エージェントと呼ばれるターゲット上のデバッグを補助するプログラムとホスト上の RIM の組合せを変えることで，相反する要求に対応できるという性質を持つ．デバッガの利用者は要求に合った RIM とエージェントの組合せを選択することで，要求を満たすデバッグ環境を構築できる．

具体的には，利用者は，メモリ制約の厳しいターゲットでは所定の機能を少ないメモリ消費で実現できる組合せを，即応性が求められるターゲットではメモリを消費して高速動作させることができる組合せを選び，交換することで，デバッグ環境を相反する 2 つの状況で利用することができる．

```
ターゲット上での記述 - (A)
    if(value == 1) { ... }

ホスト（RIM）上での記述 - (B)
    tif_get_sym(&address, "value");
    tif_get_mem(&value_on_host, address,
        sizeof(type_of_value));
    change_endian(&value_on_host):
    if(value_on_host == 1) { ... }
```

図 2 視点の違いによる記述の違い
Fig. 2 Difference in description which comes from a difference of developers' point of view.

しかし適応性のないデバッグ環境を用いた場合，デバッグ環境が利用者の要求を満たさなければデバッグ環境全体を用意しなおす必要がある．新しい機器の開発開始にともなってこれまでの使い慣れたデバッグ環境を失うとなれば，利用者にとって痛手となる．

ITRON デバッギングインタフェース仕様は上記のような特徴を持つが，RIM を作成しにくいという問題がある．以下に，RIM の作成を困難にしている 3 つの要因を示す．

(1) 記述の視点が異なる

組込み機器開発の特徴であるクロス開発環境では，プログラムが動作するコンピュータと，プログラムのデバッグを行うコンピュータが異なる．そのためデバッガに組み込まれる RIM は，ホスト上で動作するように記述しなければならない．

RIM のコードの多くはターゲット上の変数を用いて簡単な処理を行うだけであるため，作業量は少ない．しかし，ホスト上でターゲットの変数を扱うためには，いくつかの処理が必要となり，その結果，記述量が増大する．具体的な例を，**図 2** に示す．図 2 は，ターゲット上の変数を用いた条件分岐を ITRON デバッギングインタフェース仕様に対応したホスト上で動作さ

せるためのコードである．この例では，ホスト上での実行には，シンボル解決，値読み出し，エンディアン変換などの処理を行う必要があることを示している．また，ソフトウェア開発者は実行環境であるターゲット上からの視点でプログラムを把握しているため，ホスト上で動作する RIM の作成はさらに困難となる．

(2) 類似したコードの記述

ITRON デバッギングインタフェース仕様では複数の RIM とエージェントを利用することで，高い適応性を達成できる．多くの場合，デバッグ機能を構成する機能単位は，ホストとターゲットのどちらでも実行可能なコードを含んでいる．そのため，これらのコードを主にホスト上で実行するようにした RIM と，主にターゲット上で実行するようにした RIM を作成することで，異なる状況に適応可能なデバッグ環境を構築できる．

しかし (1) の問題から，同一のコードでもホストで動作する場合とターゲットで動作する場合とでは，それぞれ独立して記述しなければならない．このコードの再利用性が低さが，RIM（およびエージェント）の記述量の増大につながっている．

(3) 関連性の強いコードの分断

RIM は OS の内部構造に強く依存するため，OS に対して強い関連性を持つ．そのため，本来 RIM は OS 作成者が提供すべきとしているが，RIM の記述量が増大するにつれ，OS を作成するグループとは別に RIM を作成するグループが組織されると予想できる．すると関連性の高い 2 つのプログラムが複数のグループに分断され，保守性を欠く要因となる．具体的には，OS に加わった修正が RIM に反映されないケースや，両グループ間で仕様の解釈が異なることによるミスが生ずるケースなどがある．

上記の問題は，それぞれ密接に関連しており，問題 (1) がその他の問題の根底にある．また問題 (1) はデバッグにおいてのみの問題ではなく，ターゲット上の情報を用いてホスト上で動作するすべてのプログラムに共通の問題である．

この問題に対し，RIM の作成を容易にすることを目的に，ホスト/ターゲット処理の一体記述を提案する．ホスト/ターゲット処理の一体記述とは，関連性の強いホストとターゲットの処理を，ターゲット上における記述のまま，一体で記述するものである．いい換えると，ホスト上のプログラムである RIM を，図 2 の下側（B）のコードで記述するのではなく，上側（A）のコードで記述することを提案する．また具体的に本提案を使用した場合について，定量的な評価を行う．

2. 前　　提

本章では，本稿における前提条件を示す．

- クロス開発環境を用いる

多くの組込み機器開発は，クロス開発環境（クロスコンパイル・リモートデバッグ）を成す．同環境の特徴は，ソフトウェアを作成・検証するために用いるコンピュータ（ホスト）と，開発したソフトウェアを実行するためのコンピュータ（ターゲット）が異なる点である．この特徴は，組込み機器であるターゲットが用途に特化された専用機であることに起因する．

リモートデバッグでは，デバッグに用いているツールの種類によって，ターゲット上にターゲットの動作を制御するための支援プログラムが必要となる場合がある．本稿では，実装方法のいかんによらず，それらをまとめてデバッグエージェント（または，単にエージェント）と呼ぶ．

- ホスト負荷は無視できる

組込み機器開発において，ターゲットハードウェアはコストに直結するため，要求を満たす最小限の構成で設計され，厳しいリソース制約☆が課せられる．一方，ホストには通常 PC が利用され，ターゲットと比較すると非常に豊富なリソースを持つ．そのため，ターゲットにかかる負荷は重要視するが，ホストにかかる負荷は無視する．事実，近年の PC の高速化により，ホストの負荷がボトルネックになることはほとんどない．

- C 言語によるコーディングを仮定

組込みソフトウェアの多くは，C 言語によって記述されている．実際，文献 4) によると，全体の 8 割近い技術者が C 言語を用いて機器開発を行っている．そのため，今回対象とする言語は C 言語に限定する．

3. ホスト/ターゲット処理の一体記述の提案

3.1 目的と提案理由

本研究の目的は，RIM の作成を容易にすることである．また，本目的を達成するにあたり，我々はホスト上で動作するプログラムをターゲット上で動作するコードの形式のまま，ターゲットコードの一部として記述する方法を提案する．本手法を提案する理由として，2 つの項目をあげる．

(1) RIM は OS 作成者が提供すべきである

1 章で述べたように，RIM の作成を困難にしているものの 1 つに，人的要因によるバグの混入がある．本来，RIM の中核に相当する部分は単純なプログラム

☆ ここでいうリソースは，CPU の処理時間やメモリなどを指す．

であるにもかかわらず，作成者以外の人間が関わるため，人的要因により問題が起こりやすくなる．
この問題を回避するためには，RIM の中核に相当する部分だけでも OS 作成者自身の手により作成することが重要であると考えている．同時に，OS 作成者の負担が増大することに対して，OS 作成者にとって負担の少ない形で RIM を作成できる基盤が必要である．本手法は，その要求に合致する．

(2) 適用範囲が広い
ターゲット上の情報を用いてホスト上で動作するプログラムや，ホスト上でもターゲット上でも実行可能なプログラムの記述に関する問題は，デバッグ時のみの問題ではなく，クロス開発環境を用いる開発に共通の問題である．そのため，我々が目的としている RIM 記述量の減少以外の目的への応用も期待できると考えている．具体的には，暗号化された情報を出力するターゲット内のプログラムと，その内容を複合化するホスト上のプログラムの一体記述などがある．

上に示した 2 つの理由より，本提案は目的に対して妥当であり，また有用な手法であると判断した．近年，ハードウェアとソフトウェアの協調設計に対して一体記述が提唱されているのに対して，本提案はターゲットプログラムとホストプログラムの協調設計における一体記述と位置づけることもできる[5]．この提案により，ソフトウェア開発者によるコード記述および理解を容易にするとともに，コードの記述量を抑えることができる．さらに，関連性の強いプログラムの分離を防ぐことで，保守性を向上させる効果も期待できる．

我々は，今回提案する手法以外にも，複数の OS の RIM に共通する部分をテンプレートとして用意する方法や，デバッガとのインタフェースを拡張する方法なども検討した．しかし，テンプレートを用いる方法は本提案でも実現できる点や，インタフェースを拡張する方法は仕様本体に影響を及ぼす点などの問題点がある．よって本手法はより効果的であると判断した．

3.2 概　　要

本節では，我々が提案するホスト/ターゲット処理一体記述の概要を示す．我々が提案する手法は，関連性の強いホストとターゲットの処理を，ターゲット上における記述のまま，一体で記述するものである．

図 3 に本手法における処理の流れを示す．変換は次の手順で行われる．

(1) ユーザは，ターゲットまたはホスト上で動作する関連性の強いプログラムを，ターゲットで動作する形のまま一体で記述したソースコードを作成する．

(2) 作成したソースコードは変換プログラムによって，ターゲットで動作するコードとホストで動作するコードに分割される．このとき変換プログラムは，ターゲットで実行するコードは元の記述のまま出力するが，ホストで実行するコードにはコード生成および変換を行う．具体的には，ターゲット上のデータを利用するようなコードに対して，デバッガの機能を用いてターゲットの値の読み出しを行うように変換を加える．

(3) 変換プログラムが出力するコードは全体の一部でしかないため，ユーザは出力されたコードと残る全体を記述したコードをマージする．
コードの一部のみを一体記述の対象とするのは，アプリケーションプログラムの多くの部分はホストまたはターゲットに閉じた環境で動作するのに対し，本手法はホストとターゲットの双方に強く依存するプログラムの記述においてのみ有効であることによる．具体的にいえば，ホスト上でのみ動くコードをターゲットにおける記述の形で記述しても，コードの記述を困難にし，可読性も低下する．またターゲット上でのみ動くコードを一体記述しても，変換による効果を得られないため無意味である．また，扱うデータに関連性がないコードの一体記述も無意味である．

(4) ターゲットコードおよびホストコードをそれぞれのコンパイラによって処理し，実行可能なプログラムモジュールを得る．

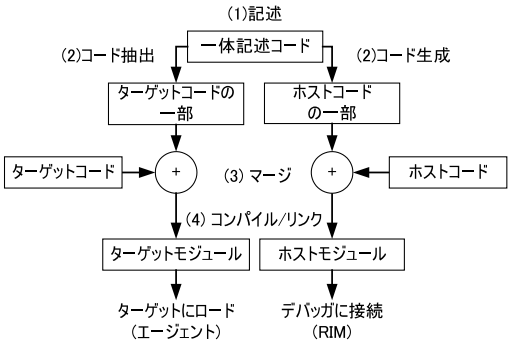

図 3 コード生成の流れ
Fig. 3 Flow of code generation.

上記の 4 つの過程を経て，ユーザは実行可能なプログラムモジュールを得る．ターゲットモジュールはターゲットコンピュータにロードし，実行される．一方，ホストモジュールはホスト内のデバッガに接続し，実行される．2 章で本研究の前提として示したように，本手法はクロス開発環境下での使用を想定している．そのため，開発環境であるホストコンピュータは次のような特徴を持つ．

- ホストはターゲット用のプログラムを生成できる．
- ホストはターゲットプログラム（ソースコード，

実行可能なプログラムモジュール，初期値，定数値）を持っている．
- ホストはターゲットのデバッグ情報（変数名，型，アドレス，関数定義など）を持っている．
- ホストはターゲットの動作を制御できる．

今回提案する手法は，ターゲットからホストへの記述変換のみに着目した．その理由として，ホストからターゲットへの記述変換は，メリットに比べデメリットが多いためである．具体的には，ターゲットはホストに比べて制約条件が多い点（リソース制約，標準ライブラリがないなど）や，組込み機器開発はターゲット主体で行われる点から，ユーザがホストからターゲットへの記述変換を利用するメリットが少ない．またターゲットはホストに比べ低速で動作するため，単純にホスト上のプログラムをターゲット上に変換しただけではオーバヘッドが大きく，実用に耐えられない可能性が高い．かつ，後述する本提案の特徴の一部が失われるというデメリットがある．

3.3 記　　述

ホスト処理とターゲット処理を一体記述した場合，両者を分離するために，いくつかの拡張を加える必要がある．本手法では，C 言語のソースコードに対し，コメントの形でいくつかの指定子を付け加える方法をとった．コメントとして記述することで，言語処理系とは独立したプログラムによる処理が容易になる．

- 実行環境指定子（host，target）
  関数または引数が格納される実行環境を指定する．ホストに格納される場合は host を，ターゲットに格納される場合は target を指定する．
- 入出力方向指定子（in，out）
  引数または変数の入出力方向を指定する．入出力方向識別子には，読み出しのみ（in），書き込みのみ（out），読み書き（inout）の 3 つが指定できる．
  関数が返却値を持つ場合，返却値は自動的に書き込み扱いとなる．また，関数がターゲットまたはホストに閉じた環境で実行される場合，入出力識別子は無効となる．
- 変数属性指定子
  C 言語で記述することによって失われた変数が持つ本来の意味を補足する．具体的には，文字列を示す string や，配列の個数を定める length_is などがある．この指定子は，CORBA で用いられている OMG IDL を参考にした[6]．

図 4 に，文字列出力を行う puts 関数の記述の例を示す．図の puts は，ターゲットから文字列の引数を受け，ホスト上で実行されることを示す記述が追加されている．

```
/* [host] */
void puts
(/*[target,in,string]*/ const char * msg)
{
    while(*msg != '\x0')
        putc(*(msg++));
    putc('\n');
}
void main(void)
{
    const char * letter;
    puts("Hello, the world!!");
    letter = "Hello, the world!!";
    puts(letter);
}
```

図 4 puts 関数の記述
Fig. 4 Description of 'puts'.

3.4 特　　徴

ターゲット側のコードからホスト上のコードを呼び出すという点に関しては，本手法は広く知られている Remote Procedure Call（以下，RPC）と変わりはない．しかし本手法は以下に示す項目において，RPC と決定的に異なる．

- ターゲット上での記述を用いる点

通常 RPC などでは，サーバプログラムはサーバコンピュータ上での記述で作成され，クライアントプログラムはクライアントコンピュータ上での記述で作成される．またサーバ/クライアント間で，引数や返却値の送受は行えるが，サーバ上の大域変数を利用することはできない．

これに対し本手法では，ホスト上で動作するプログラムの記述も，ターゲット上での記述の形で提供する点が新しい．具体的には，図 4 ではターゲット上で putc 関数を用意すれば，そのままのターゲットプログラムとして処理することも可能である．

- 専用のプロキシ/スタブが不要である点

RPC では，データのマーシャリングや転送などを行うスタブと呼ばれるプログラムが必要となる．かつ，スタブはサーバ側およびクライアント側それぞれに必要となる．これは，サーバとクライアントの間に高い独立性があるためである．

一方本手法では，ホストのみがスタブを持つ．これは，クロス開発環境を前提としているため，ホストはター

ゲットのデバッガであることに起因する．ターゲットはホストの資源を利用することはできないが，ホストはデバッガであるため，ターゲットの任意の資源を操作可能である．そのため，専用のターゲットスタブは不要となり，デバッグエージェントで代用できる．特にターゲットのリソース制約が厳しい場合，専用のターゲットスタブが不要となる効果は大きい．

また本手法を ITRON デバッギングインタフェース仕様のようなデバッグインタフェース上で利用することで，同仕様に対応したさまざまなツールで用いることができ，変換ツールは高い汎用性を確保できる．

4. 組込み機器のデバッグへの適用と評価

本章では，前章で示した提案を実際のデバッグに適用し，具体的な例を示すとともにその評価を行う．

デバッグ作業において，文字列の出力はターゲットの状態を知るための手段として広く用いられている．そのため，文字列出力関数（puts）について一体記述を行った．記述した内容を図 4 に示す．

評価対象として puts 関数を選んだ 3 つの理由を以下に示す．

(1) デバッグ時に広く利用される機能であること
本手法が実際のデバッグ時に利用できることを示すために，対象となる関数もデバッグ時に広く使われている関数が望ましいと判断した．これに対し，puts 関数は LED に並んでプログラムの実行位置を確認するために広く用いられており，妥当であると判断した．

(2) 軽量であること
プログラムの動作検証などを行う場合，ターゲットの動作に大きく影響を及ぼすような大規模な関数を利用すると，システム全体の振舞いが変わってしまう恐れがある．そのため，デバッグには比較的軽量な関数を用いるのが一般的である．puts 関数は関数としてそれほど大きくなく，与える文字列の長さが予測可能であればシステムに与える影響も予測可能であるという特徴を持ち，デバッグ時には広く用いられると判断した．

(3) その他のデバッグ機能の基底機能であること
手法の検証を行うために，デバッグ時に用いられる多くの関数に共通した機能を持っていることが望ましい．このような関数を選定することで，単一の関数の機能の評価によって，他の関数に適用した結果を類推することが容易になる．puts 関数は，ターゲットからホストへの情報の転送という基本的な機能を持っているため，本手法の第 1 段階の評価としてふさわしいと判断した．またリモートデバッグ時には機器間通信は無視できないオーバヘッドとなるため，評価項目として重要であると判断した．

本評価によって，ターゲットからホストへの情報の転送という機能に着目して，ターゲット上の変数を引数に持つ関数が一体記述可能であることと，その一体記述からホスト上で動作する関数が作成可能であること（適応性評価）を示す．

4.1 評価環境

今回の評価で利用したターゲットは，日立製作所製 SolutionEngine MS7709ASE01（SH7709A プロセッサ，内部クロック 133 MHz，ライトスルーキャッシュ）である．コンパイラには GNU C Compiler（以下，gcc）を利用し，-O2 で最適化を行っている．デバッガには ITRON デバッギングインタフェース仕様に対応した GNU Debugger（以下，gdb）を利用している[2]．また次のような独自拡張を含んでいる．

- 式の評価（vevl_exp）
  gdb が解決可能な任意の式を与えると，式を解決した結果を返却する．

ターゲット上には gdb デバッグエージェント（通称スタブ[7]）が組み込まれている．スタブはターゲットごとに作成され，メモリ読み出し，書き込み，実行制御など単純なターゲット制御を行うデバッグエージェントである．gdb は，ターゲット（スタブ）とホストの間の通信をシリアル通信路（115.2 kbps）経由で行う．

4.2 実　　装

図 4 で示した puts 関数をホスト上で動作させるため，変換プログラムは次のような変換を行う．

(1) ターゲット用 puts 関数の作成
ターゲットコードに出力される puts 関数は，中身が空の状態で出力される．変換プログラムは，ターゲットコード内に中身が空の puts 関数を出力する．しかし真に中身がない関数を作成すると gcc が最適化により関数呼び出し自体を削除してしまうため，引数を取得することができない．そのため，インラインアセンブラを用い，最適化によってコードが排除されないような記述がなされている．

(2) ホスト用 puts 関数ラッパーの作成
今回の puts 関数は引数に target 指定子があるため，ラッパー関数が生成される．ラッパー関数では，puts 関数に渡す引数をターゲットから読み出す処理を行う．また返却値などをターゲット上に書き込む処理も行う．ラッパー関数は，ターゲットの現在実行番地からどの puts が呼ばれたのかを判断するための変換テーブルを持つ．変換テーブルは puts 関数の引数部が格納されている．具体的には，図 4 の場合，最初の puts 関数に対しては，”Hello the world!!”という文字列が登

録されている．続く puts 関数に対しては，letter という変数名が登録されている．ラッパー関数はこれらの情報を元に，ターゲット上から情報を取り込む．

一体記述したファイル以外からの呼び出しにも対応するため，変換テーブルはデフォルトの引数としてレジスタ名 R4 を格納する．R4 は SH プロセッサ用 gcc の第 1 引数格納レジスタに相当する．

今回の puts 関数は返却値を持たないが，返却値を持つ関数のラッパーは関数の返却値をターゲットの R0 レジスタに格納する．R0 は SH プロセッサ用 gcc の返却値格納レジスタに相当する．

(3) 文字列定数を引数に持つ puts の削除

変換プログラムは同一ファイル内の puts 関数の呼び出しをすべて削除し，アドレスを参照するためのシンボル定義に置き換える．

同一ファイル内の puts 関数を削除するのは，ホストは変換テーブルによってどの位置でどのような引数をともなって puts 関数が呼ばれているかを把握しているため，引数の格納自体が不要であることに起因する．

本来は，関連性の高いコードを一体記述することが目的であるため，一体記述されていないファイルから対象となる関数が呼び出されることは望ましくない．しかし，現実には起こりうる操作であると判断し，評価を目的に上記のような形で対応した．

なお，ターゲット上で動作させる puts 関数を作成する場合（puts 関数に target 指定子を指定した場合），変換プログラムは何も行わない．

ターゲットを実行させると，変換プログラムが生成したホストプログラムは次のような動作を行う．

(1) 変換テーブルに登録されている全 puts 関数の位置にブレークポイントを設置する．

(2) ターゲットの実行がブレークポイントにより停止すると，現在のターゲットの実行位置を取得し，変換テーブルに登録されているかどうかを確認する．

(3) 登録されていた場合，生成したホスト用 puts 関数ラッパーを呼び出し，ラッパーはホスト用 puts 関数を呼び出す．

(4) ターゲットの実行を再開させる．

今回，ブレークポイントの設置やブレークヒットのフックなどのデバッガ依存の処理を行うにあたり，ITRON デバッギングインタフェース仕様のターゲットアクセスインタフェース関数群を利用している．

図 5 に，図 4 の一体記述コードから上記の手法で生成したターゲットコードを示す．図 4 に含まれる puts 関数には host という指定があるため，関数はホスト上で実現される．そのため，ターゲット上のコードの中身は空となるが，最適化によって削除されるのを防ぐために，やむなく無意味なインラインアセンブラを使用している．関数は inline を用いてインライン展開を行うことで，無用なコードの生成を抑えている．inline は多くの処理系で実装されている点，および C99 で標準に採用されている点から使用しても問題ないと判断した．

```
inline void puts(const char * msg)
{ __asm("");  }
void main(void)
{
   const char * letter;
_PUTS_1: /* puts("Hello, the world!!"); */
   letter = "Hello, the world!!";
_PUTS_2: /* puts(letter); */
}
```

図 5 puts の変換結果（ターゲットコード）
Fig. 5 Translation result of puts (target side).

```
void puts(const char * msg)
{
   while(*msg != '\x0')
       putc(*(msg++));
   putc('\n');
}
void wrapper_puts(ADDRESS PC)
{
   char * msg;
   msg = load_string(get_argument(PC,0));
   puts(msg);
   do_cleanup();
}
```

図 6 puts の変換結果（ホストコード）
Fig. 6 Translation result of puts (host side).

図 6 に，図 4 の一体記述コードから上記の手法で生成したホストコードを示す．本体である puts 関数には，保守性および可読性を高めるためにできる限り修正を加えないようにした．get_argument 関数は，ターゲットの実行位置から関数の引数を得る関数である．具体的には，PC=__PUTS_1 であった場合には，文字列定数を格納する ROM 上のアドレスを返す．PC=__PUTS_2 であった場合には，引数である letter のアドレスを返す．一体記述コード外からの呼び出しであった場合には，引数レジスタ（R4）の値を返す．load_string 関数は，ターゲットアドレスから文字列を

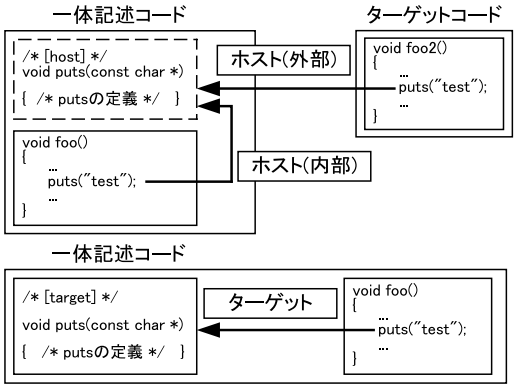

図 7 3 種類の puts 関数
Fig. 7 Three implementations of puts function.

得る関数である．アドレスが RAM 空間であった場合，ターゲット上から文字列を読み出す．一方 ROM 空間内であった場合，デバッガが持つロードモジュールより文字列を得る．do_cleanup 関数は，load_string 関数などが確保した動的なメモリ領域を解放する．

4.3 記述量の低下に関する評価

本節では，一体記述導入による RIM 記述量削減効果の評価を行い，目的の達成を確認する．なお，2 章の前提 (2) により，ホストコンピュータにかかる負荷は評価から外している．

評価では，4.2 節で示した変換プログラムを用い，図 4 の一体記述から次の 3 種類の puts 関数を作成した．

- ホスト（内部）
- ホスト（外部）
- ターゲット

4.2 節で示したように，ホスト上で動作する puts 関数は 2 種類ある．ホスト（内部）は，同一記述ファイル内からの呼び出し時に利用される puts 関数であり，本提案における標準的な puts 関数の実装となる．一方，ホスト（外部）は，同一記述ファイル外からの呼び出し時に利用される puts 関数である．引数の格納など，いくつか省略不可能なオーバヘッドを含んでおり，ホスト上で動作する現実的な最悪ケースとなる．ターゲットは，変換を加えずターゲットコンパイラで生成した puts 関数である．それぞれの関数呼び出しの関係を，**図 7** に示す．

一体記述を行うことによってどの程度記述量が減少したかを示すために，記述量比率を用いる．記述量比率は，変換プログラムによって生成されたプログラムの行数に対するユーザ記述行数の比率とする．記述量比率は低ければ低いほど良い．値 1.0 は変換の前後で記述量が変化しないことを示す．

表 1 コード行数
Table 1 An amount of target program (in lines).

項目		コード行数
ユーザ記述		7 行
ホスト（内部）	ターゲット側	0 行
	ホスト側	55 行
	記述量比率	0.13
ホスト（外部）	ターゲット側	4 行
	ホスト側	55 行
	記述量比率	0.12
ターゲット	ターゲット側	7 行
	ホスト側	0 行
	記述量比率	1.0

表 1 にコード行数を示す．項目のターゲット側およびホスト側とは，変換によって生成されたプログラムのうち，ターゲットで動作する部分とホストで動作する部分を指している．

ターゲットとホスト変数およびホスト定数の間では，プログラムの記述量は約 13%に減少した．またターゲットで動作させる場合においては，一体記述のために追加したいくつかの記述が無駄となるが，その量はそれほど多くなく，特に問題はないと判断している．また実際にプログラムを実行させ，引数にターゲット上の値を使用しホスト上で動作するようなプログラムが，一体記述から生成可能であることも確認した．

この結果より，本手法は RIM の記述量を削減するという目的を達成している．また記述が容易になるという点に関してであるが，定量的な評価はできないが，図 4 で示したプログラムは gdb 内部の関数や ITRON デバッギングインタフェース仕様の関数などに関する記述を含まないため，容易に理解できるものと判断できる．

4.4 適応性評価

本稿で述べる適応性とは，RIM およびエージェントを交換することでデバッガの特性を変化させ，利用者の環境や状況に適応できる性能を指す．そのため本節では，一体記述された puts 関数の記述から複数の RIM とエージェントの組を生成し，デバッガがこれらのモジュールの組を使い分けることで異なる特性を持つことを確認することで，生成したコードがデバッガに適応性を与えることを示す．

1 章の問題 (2) で示したような状況では，一体記述を行うことにより異なる 2 つの性質を持った RIM を単一の一体記述から生成可能である．そこで，本手法を用いた場合に得られる適応性の指標として，一体記述から生成した RIM が持つ性質を定量的に評価する．なお，2 章の (2) により，ホストコンピュータにかか

る負荷は，評価から外している．

適応対象となる状況として，次の 2 つを想定する．

- 使用できるメモリに制限があるような状況
- 機器間通信路が非常に低速であるような状況

前者にはデバッグに要するメモリが 1 バイトでも少ない方がよいという要求がある☆．後者には通信が 1 バイトでも少ない方が良いという要求がある．通常の場合，両者は相反関係にあるため☆☆，2 つの要求を同時に満たすデバッガを得るのは困難である．そのため，モジュールの交換のみで 2 つの要求を満たすことができるならば，モジュールはデバッガに適応性を与えているということができる．

この要求におけるデバッガの適応性を示す定量的な指標として，次の 2 項目を用いる．

- ターゲットプログラムサイズ
- 実行速度（通信データ量）

ターゲットプログラムサイズは，変換プログラムによって生成されたターゲットプログラムのサイズである．このサイズには，呼び出し元関数が関数を呼び出すために作成するコード（フレーム生成など）も含める．実行速度には，通信データ量を元に算出した値を用いる．通信データ量は，バイト数とパケット数で示す．ここでいうパケット数とは，送信および受信の合計回数に相当する．

実行速度に実測値を用いない理由は，ホストコンピュータやターゲットコンピュータの処理速度に比べ通信速度が非常に遅く，通信がプログラム実行速度の大部分を支配しており，予測値と実測値の間にそれほど大きな差が生じないためである．またパケット数を示すのは，Ethernet を通信路に用いるようなデバッグツールでは，総転送バイト数よりもパケット数のほうが通信時間に及ぼす影響が大きいためである．

評価では，ホスト上で動作する puts 関数との比較を行いやすくするため，ターゲット上に 2 種類の putc 関数を用意した．1 つは，シリアルインタフェースに文字を直接出力する putc 関数（以下，ターゲット（シリアル））である．もう 1 つは，gdb の文字出力機能を利用した putc 関数（以下，ターゲット（gdb））である．記述した puts 関数の目的が文字列を出力することであった場合，前者の putc 関数を用いることで高速に動作する．しかし，gdb が用いているシリアル通信路に加え，文字列出力専用の通信路が必要となる．一方，目的がデバッガの画面上に文字列を出力することであった場合，後者の putc 関数を用いる必要がある．これは，今回提案している一体記述を用いることなくホスト（外部）と同じ処理を行おうとした場合に，一般的に用いられている文字列出力方法である．

表 2 文字列出力時の通信データ量を示すパラメータ
Table 2 Coefficients that describes an amount of communication.

項目	定数項 (c)	係数 (a)
ターゲット （シリアル）	0 (1)	1 (0)
ターゲット （gdb）	0 (0)	8 (2)
ホスト （外部）	35 (4)	6 (0.5)
ホスト （内部）	35 (4)	0 (0)

表 3 18 文字出力時の通信データ量
Table 3 An amount of communication with 18 words.

項目	総通信量	所要時間
ターゲット （シリアル）	18 (1)	1.3 ms
ホスト （内部）	35 (4)	2.4 ms
ホスト （外部）	143 (13)	9.9 ms
ターゲット （gdb）	144 (36)	10.0 ms

表 4 ターゲットプログラムサイズ
Table 4 An amount of target program size.

項目	プログラムサイズ
ホスト（内部）	0 バイト
ホスト（外部）	14 バイト
ターゲット	46 バイト

表 2 に，各項目ごとの通信データ量を示す係数および定数項を示す．今回のケースでは，ホストとターゲットの間での通信量は文字列の長さに対して比例関係（総通信量 = 係数 (a) × 文字列の長さ + 定数項 (c)）にあるため，表 2 より任意の長さの文字列に対する結果が計算可能である．係数は 1 文字の出力に要する通信量（文字の送信）を意味し，定数項は 1 回の文字列出力操作に要する基礎的な通信量（ブレーク通知，PC 取得など）を意味する．

また 18 文字の文字列を転送した結果を**表 3** に示す．数値はそれぞれバイト数を示し，括弧内はパケット数を示す．一部端数となっている部分は，gdb が 1 回のトランザクションで 4 文字を同時に取得することに起因する．表 3 の所要時間は通信バイト数より算出したものである．詳しくは 4.1 節を参照されたい．

また，**表 4** に，変換したプログラムのうち，ターゲットプログラムのサイズを示す．なお，putc 関数のサイズはターゲットプログラムサイズに含まれない．

☆ 今回使用した SH7709A は比較的多くのメモリを持っているが，アプリケーションによってはデバッグに利用できるメモリがわずかしか残っていないケースもある．

☆☆ メモリ消費の削減には処理をホストで行うことが有効であるが，処理のために機器間の通信が増える．通信の削減には内容の圧縮が有効であるがプログラムを要するためメモリを圧迫する．

3つの表が示す結果より，一体記述からホスト上で動作するように作成したコードは，ターゲットメモリ消費を低くするという点で有利であるという結果を得た．特に，同一記述ファイル内から puts 関数を呼び出している場合，ターゲット上にはいっさいの追加プログラムが不要となる．この特徴は，リソース制約の厳しい機器のデバッグに効果的である．

同一記述ファイル外から puts 関数を呼び出している場合，空の関数定義に 4 行を要し，gcc が生成する関数呼び出しコードによりターゲット上の ROM を 14 バイト消費する．このオーバヘッドは puts 関数を呼び出すたびに発生し，プログラム中に 100 カ所呼び出しがあれば，ターゲット上の ROM を 1400 バイト消費する．

一方，一体記述をそのままターゲット上で実行した場合，シリアル通信路を直接利用することで，実行速度を高速化できるという点で有利であるという結果を得た．この場合でも，ホスト（外部）と同様に，関数呼び出しコードのため呼び出しごとに 14 バイトのターゲット ROM を消費する．しかし，文字列定数を出力する場合でもすべての文字列をホストに転送するという点で，ホスト（外部）と異なる．

同一の動作を行うホスト（外部）とターゲット（gdb）に着目すると，表 2 より文字列の長さが 18 文字未満であるときはターゲット（gdb）が高速であるが，18 文字以上であるときはホスト（外部）の方が高速となることが分かる．また表 4 から，ホスト（外部）はターゲット（gdb）に比べ，ターゲット上におけるメモリ消費量が約 70%削減されている．

上記の結果より，同一記述からメモリ消費量を抑えた実装と実行速度を重視した実装の 2 種類のコードが作成できることを確認した．このことより，文字列の出力という機能に限ってではあるが，デバッガは本手法によって作成したコードを用いることで，2 つの異なる特性を持つことを確認した．よって生成したコードはデバッガに適応性を与えているといえる．

ここでの評価は文字列出力に限定したものであるが，本手法はデバッグ時のみに必要な機能の多くに適用することが可能であると考えている．その理由として，デバッグ情報は人間が認識しやすいように冗長である場合が多く，また定数（特に文字列定数）を多く含んでいる点がある．このようなデバッグ時のみに必要な定数を，ターゲット上ではなくホスト上に配置するようにすることで，メモリ消費量およびターゲットとホストの通信量を抑えることができる．本稿で定量的な評価結果を示すまでに至らなかったが，assert 関数☆や，printf 関数でも同様の結果を得ている．ただし，今回はシリアル通信路を用いたためにこのような結果になったが，ICE や JTAG などの機器を用いた場合では，同様の傾向は得られるものの，実行時間の差は少なくなると予想している．

本評価から，4.2 節で示した変換によって，ターゲット上の記述をホスト上で実行できるという結果を得た．このことは，1 章の問題 (1) は本手法により解決できることを示している．また，同一のコードから性質の異なる複数の RIM が作成可能であるという結果から，1 章の問題 (2) に対しても本手法は効果的であることを示している．

5. ま と め

本稿で我々は，RIM の記述を容易にすることを目的に，ホスト/ターゲット処理の一体記述を提案した．またホスト/ターゲットの一体記述が実際に有用であることを示すため，puts 関数を例にあげ，一体記述からホスト上で動作するプログラムモジュールが生成できることを確認した．また，記述量の減少を定量的に評価するとともに，記述が容易になることを定性的に示した．また同時に，単一の puts 関数の一体記述から，異なる性質を持った複数の puts 関数を作成可能であることも確認した．

今後の課題として，より詳細な記述の妥当性に関する評価がある．今回文字列定数を引数に持つ puts 関数は削除することとしたが，削除してはいけない場合もありうる．そのような場合，volatile 指定子を導入する方法が有効であることが分かっている．しかし，まだ見つかっていない記述上の問題があると考えている．また，記述子の記述方法に関してもより細かい検討が必要であると考えている．

また，具体的な実装方法に関しての評価が不十分である．今回は puts 関数を例にとり，ターゲット上の変数を引数に持つような場合を示したが，関数内でターゲット上の大域変数を扱う場合や，ポインタ操作を行った場合における評価が欠けている．これらに関しては，実装方法の目処はついており，実際に実装できると考えているが，評価を行うには至っていない．

また，ホスト側の記述に C++言語を用いることも検討している．変換プログラムが出力するホストコードが C++になることで演算子オーバロードが利用でき，それほど高度な変換プログラムを用いなくても同

☆ 条件が偽であるとプログラムの実行を中断する関数．不正状態の検出などに用いられ，中断時にデバッグに必要な情報（ファイル名や条件式など）を出力するように改造されていることが多い．

様の処理を行うことが可能であると考えている．組込み分野においてはC++言語はオーバヘッドなどの問題からいまだ一般的とはいえないが，対象はホスト上で動作するコードであるため，C++言語の使用はそれほど問題にならないと考えている．

参 考 文 献

1) Barr, M.: *Programming Embedded Systems in C and C++*, O'REILLY (1999). 有馬三郎（訳）：C/C++による組み込みシステムプログラミング，オライリージャパン (2000).
2) 若林隆行，高田広章：ITRONデバッギングインタフェース仕様における標準化アプローチとその適応可能性に関する評価，情報処理学会論文誌，Vol.42, No.6, pp.1503–1513 (2001).
3) トロン協会 ITRON 部会：ITRON デバッギングインターフェース仕様 ver 1.00.00，トロン協会 (2000).
4) 社団法人トロン協会：ITRONプロジェクトホームページ．http://www.itron.gr.jp/.
5) Gajski, D.D., Zhu, J., Dömer, R., Gerstlauer, A. and Zhao, S.: *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers (2000).
6) OMG, I.: *The Common Object Request Broker: Architecture and Specification revision 2.5*, Object Management Group, Inc. (2001).
7) Stallman, R.M. and Pesch, R.H.: *Debugging with GDB fifth edition*, Free Software Foundation (1998). コスモプラネット（訳）：GDBデバッギング入門，アスキー出版 (1999).

(平成13年12月17日受付)
(平成14年 4 月16日採録)

若林 隆行

2001年豊橋技術科学大学大学院情報工学専攻修士課程修了．現在，同大学院電子・情報工学専攻に在学．リアルタイムOS，デバッグ技法，ソフトウェア開発環境の研究に従事．修士（工学）．

高田 広章（正会員）

豊橋技術科学大学情報工学系助教授．1988年東京大学大学院理学系研究科情報科学専攻修士課程修了．同学科の助手等を経て，1997年12月より現職．リアルタイムOS，リアルタイムスケジューリング理論，組込みシステム開発技術等の研究に従事．ITRON仕様の標準化活動に，中心的メンバとして参加．博士（理学）．IEEE，ACM，電子情報通信学会，日本ソフトウェア科学会各会員．