OpenVMS
デバッガ説明書

AA–PY9GF–TE.2

2010 年 10 月

本書は，高級言語とアセンブリ言語のプログラマのための OpenVMS デバッガの機能について説明します。

改訂／更新情報: OpenVMS V8.2 『デバッガ説明書』の改訂版です。
ソフトウェア・バージョン: OpenVMS Integrity Version 8.4
OpenVMS Alpha Version 8.4

日本ヒューレット・パッカード株式会社

原典： OpenVMS Debugger Manual

本書は，日本語 VAX DOCUMENT V 2.1を用いて作成しています。

目次

C 各言語に対するデバッガ・サポートの要約

索引

Example

図

表

まえがき

対象読者

本書は，デバッガを使用するすべてのプログラマを対象とします。本書には，デバッガの次の 2 つのユーザ・インタフェースについての内容が含まれています。

- 端末およびワークステーションで使用するコマンド・インタフェース
- ワークステーションで使用する DECwindows Motif for OpenVMS ユーザ・インタフェース
- Microsoft Windows PC クライアント・インタフェース

OpenVMS Integrity あるいは OpenVMS Alpha システムの OpenVMS デバッガを使用すると， OpenVMS オペレーティング・システムの 64 ビット処理により使用可能になる，すべての拡張メモリにアクセスできるようになります。このため，完全な 64 ビット・アドレス空間でデータのテストと処理が行えるようになります。

OpenVMS デバッガはあらゆる地域で使用できるよう設計されています。アジア地域のユーザであれば，デバッガの DECwindows Motif for OpenVMS，コマンド行，画面モード・ユーザ・インタフェースをマルチバイト文字で使用することもできます。

デバッガを使用してコードをデバッグすることができるのは，ユーザ・モードの場合だけです。スーパバイザ・モード，エグゼクティブ・モード，カーネル・モードではコードをデバッグすることはできません。

本書の構成

本書は次の章と付録で構成されています。

- 第 1 部では， OpenVMS デバッガを紹介します。第 1 部には章が 1 つあります。
  - 第 1 章は，デバッガの概要を説明します。
- 第 2 部では，デバッガのコマンド・インタフェースについて説明します。第 2 部には次の章があります。
  - 第 2 章では，デバッガの使用を開始するための事項について説明します。
  - 第 3 章では，プログラムの実行を制御およびモニタする方法について説明します。
  - 第 4 章では，プログラム・データの検査と操作の方法について説明します。
  - 第 5 章では，プログラム内のシンボルへのアクセスを制御する方法について説明します。
  - 第 6 章では，ソース・コードの表示を制御する方法について説明します。
  - 第 7 章では，画面モードの使用方法について説明します。
- 第 3 部では，デバッガの DECwindows Motif for OpenVMS ユーザ・インタフェースについて説明します。第 3 部には次の章があります。
  - 第 8 章では，デバッガについて紹介し，その DECwindows Motif for OpenVMS ユーザ・インタフェースの機能についての概要を示します。
  - 第 9 章では，デバッグのためのプログラムの準備方法と，それから DECwindows Motif for OpenVMS ユーザ・インタフェースを使用してのデバッグ・セッションの開始と終了の方法について説明します。
  - 第 10 章では， DECwindows Motif for OpenVMS ユーザ・インタフェースによってのデバッガの使用方法を各タスクごとに説明します。
- 第 4 部では，デバッガの PC インタフェースについて説明します。第 4 部には章が 1 つあります。
  - 第 11 章では，デバッガの PC インタフェースの概要を説明します。
- 第 5 部では，高度なデバッガのトピックについて説明します。第 5 部には次の章があります。
  - 第 12 章では，デバッガのヒープ・アナライザの使用方法を各タスクごとに説明します。
  - 第 13 章では，キー定義やその他のカスタマイズなどの便利な追加機能について説明します。
  - 第 14 章では，最適化されたプログラムや複数言語プログラムなどの特殊な事例について説明します。
  - 第 15 章では，マルチプロセス・プログラムのデバッグ方法について説明します。

– 第 16 章では，タスキング・プログラム (マルチスレッド・プログラム) のデバッグ方法について説明します。

- 第 6 部には次の章があります。
  – 付録 A では，デバッガによってあらかじめ定義されているキーパッド・キー定義の一覧を示します。
  – 付録 B では，デバッガのすべての組み込みシンボルと論理名を示します。
  – 付録 C では，デバッガによる各言語サポートを示します。
  – 付録 D では，第 8 章，第 9 章，および，第 10 章の図で紹介されているプログラムのソース・コードを示します。

関連資料

デバッガを使用する際には，次の資料も参考になります。

プログラミング言語
本書では，デバッガでサポートしている言語の大部分に共通する使用法について記述しています。特定の言語に固有の情報についての詳しい説明は，次の資料を参照してください。

- デバッガのオンライン・ヘルプ・システム (第 2.1 節を参照)
- 各言語に提供されている資料のうち，特にプログラムのデバッグのためのコンパイルとリンクに関する資料
- VAX アセンブリ言語命令とVAX MACROアセンブラの詳細について説明している『VAX MACRO and Instruction Set Reference Manual』または『MACRO–64 Assembler for OpenVMS AXP Systems Reference Manual』

◆リンカ・ユーティリティ
プログラムや共用可能イメージのリンクについての詳しい説明は，『OpenVMS Linker Utility Manual』を参照してください。

◆ Delta/XDelta デバッガ
スーパバイザ・モード，エグゼクティブ・モード，カーネル・モード (つまり，ユーザ・モード以外のモード) でのコードのデバッグについての詳しい説明は，ドキュメント・セットの『OpenVMS Delta/XDelta Debugger Manual』を参照してください。このマニュアルには，特権プロセッサ・モードで実行するプログラムや，高い割り込み優先順位で実行するプログラムのデバッグについての情報が記載されています。

◆ OpenVMS System-Code デバッガ
オペレーティング・システム・コードのデバッグについては，『 OpenVMS System Analysis Tools Manual』を参照してください。このマニュアルには，OpenVMS デバッガから OpenVMS System-Code デバッガを起動する方法，OpenVMS System-Code デバッガ環境でのデバッグ方法についての情報が記載されています。

OpenVMS System-Code デバッガ固有のコマンドについての詳しい説明は，『デバッガ・コマンド・ディクショナリ』の CONNECT コマンドと REBOOT コマンドの項を参照してください。

◆ DECwindows Motif for OpenVMS

DECwindows Motif for OpenVMS ユーザ・インタフェースの一般的な情報については，『VMS DECwindows User's Guide』を参照してください。

この他の OpenVMS 製品やサービスについての詳細は，次の Web サイトを参照してください。

https://www.hpe.com/info/openvms

本書で使用する表記法

製品名について

VMScluster システムは， OpenVMS クラスタ・システムを指します。

また，日本語 DECwindows および日本語 DECwindows Motif はすべて日本語 DECwindows Motif for OpenVMS ソフトウェアを意味します。

例について

本書には，デバッガの DECwindows Motif ユーザ・インタフェースを示す図が多数収録されています。このインタフェースの画面構成はそれぞれのユーザごとにカスタマイズできるため，ユーザのシステム上のデバッガ表示と一致しないことがあります。

OpenVMS Integrity あるいは OpenVMS Alpha システムの OpenVMS デバッガは，OpenVMS オペレーティング・システムの 64 ビット処理により使用可能になるすべての拡張メモリに対してアクセスできるようになっていますが，本書のサンプルは，その事実を反映するよう更新されてはいません。そのため， 16 進アドレスは，Alpha では 16 桁の数になります。つまり次の例のようになります。

```
DBG> EVALUATE/ADDRESS/HEX %hex 000004A0
00000000000004A0
DBG>
```

また，本書では，次の表記法を使用しています。

表記法	意味
Ctrl/x	Ctrl/xという表記は， Ctrl キーを押しながら別のキーまたはポインティング・デバイス・ボタンを押すことを示します。
PF1 x	PF1 xという表記は， PF1 に定義されたキーを押してから，別のキーまたはポインティング・デバイス・ボタンを押すことを示します。

表記法	意味
Return	例の中で，キー名が四角で囲まれている場合には，キーボード上でそのキーを押すことを示します。テキストの中では，キー名は四角で囲まれていません。 HTML 形式のドキュメントでは，キー名は四角ではなく，括弧で囲まれています。
. . .	例の中の水平方向の反復記号は，次のいずれかを示します。 • 文中のオプションの引数が省略されている。 • 前出の 1 つまたは複数の項目を繰り返すことができる。 • パラメータや値などの情報をさらに入力できる。
. . .	垂直方向の反復記号は，コードの例やコマンド形式の中の項目が省略されていることを示します。このように項目が省略されるのは，その項目が説明している内容にとって重要ではないからです。
( )	コマンドの形式の説明において，括弧は，複数のオプションを選択した場合に，選択したオプションを括弧で囲まなければならないことを示しています。
[ ]	コマンドの形式の説明において，大括弧で囲まれた要素は任意のオプションです。オプションをすべて選択しても，いずれか 1 つを選択しても，あるいは 1 つも選択しなくても構いません。ただし，OpenVMS ファイル指定のディレクトリ名の構文や，割り当て文の部分文字列指定の構文の中では，大括弧に囲まれた要素は省略できません。
[|]	コマンド形式の説明では，括弧内の要素を分けている垂直棒線はオプションを 1 つまたは複数選択するか，または何も選択しないことを意味します。
{ }	コマンドの形式の説明において，中括弧で囲まれた要素は必須オプションです。いずれか 1 つのオプションを指定しなければなりません。
太字	太字のテキストは，新しい用語，引数，属性，条件を示しています。
italic text	イタリック体のテキストは，重要な情報を示します。また，システム・メッセージ (たとえば内部エラー*number*)，コマンド行 (たとえば /PRODUCER=*name*)，コマンド・パラメータ (たとえば*device-name*) などの変数を示す場合にも使用されます。
UPPERCASE TEXT	英大文字のテキストは，コマンド，ルーチン名，ファイル名，ファイル保護コード名，システム特権の短縮形を示します。
`Monospace type`	モノスペース・タイプの文字は，コード例および会話型の画面表示を示します。 C プログラミング言語では，テキスト中のモノスペース・タイプの文字は，キーワード，別々にコンパイルされた外部関数およびファイルの名前，構文の要約，または例に示される変数または識別子への参照などを示します。
-	コマンド形式の記述の最後，コマンド行，コード・ラインにおいて，ハイフンは，要求に対する引数がその後の行に続くことを示します。
数字	特に明記しない限り，本文中の数字はすべて 10 進数です。 10 進数以外 (2 進数， 8 進数， 16 進数) は，その旨を明記してあります。

第1部
デバッガ概要

第１部では，デバッガの概要を説明します。

1
デバッガ概要

本章では OpenVMS デバッガのコマンド・インタフェースについて説明します。本章には次の内容が含まれています。

- デバッガの機能についての概要
- デバッグ用プログラムのコンパイルとリンクの方法
- デバッガ・セッションの開始と終了の方法
- 機能別に分けられたデバッガ・コマンドの一覧

基本的なデバッグ・タスクについては，第 2 章を参照してください。

1.1 デバッガの概要

OpenVMS デバッガは，実行時のプログラミング・エラーや論理エラーなどのバグの場所をつきとめるためのツールです。コンパイルとリンクには成功しても正常に実行されないプログラムに対してデバッガを使用します。このようなプログラムは，たとえば，不正な出力を行ったり，無限ループに陥ったり，途中で終了してしまいます。

OpenVMS デバッガを使用すると，プログラムを実行しながら会話形式でプログラムの動作をモニタし，操作することにより，プログラムのエラーを見つけることができます。デバッガ・コマンドを使用すると，次のことが可能です。

- プログラムの実行を制御し，監視できる。
- プログラムのソース・コードを表示し，ブラウズして，注意すべき命令と変数を細かく確認できる。
- 指定したポイントでプログラムの実行を中断して，変数やプログラムの他の要素の変化をモニタできる。
- 変数の値を変更する。場合によっては，ソース・コードの編集や再コンパイルおよび再リンクを行わずに，変更したプログラムをテストできる場合もある。
- プログラムの実行パスをトレースする。
- プログラムの実行中に変数やその他のプログラム要素の変化をモニタする。

以上が基本的なデバッグ方法です。プログラム内のエラーを特定することができれば，ソース・コードを編集，コンパイル，リンクして，修正バージョンを実行することができます。

デバッガとそのドキュメントを使用していくうちに，デバッグには基本的な方法以外にさまざまな方法もあることが分かります。また，ユーザのニーズに合わせてデバッガをカスタマイズすることもできます。第 1.1.1 項では， OpenVMS デバッガの機能の概要を説明します。

1.1.1 デバッガの機能

◆サポートしているプログラミング言語

Alpha プロセッサでデバッガを使用する場合，次の各言語で記述されたプログラムをデバッグすることができます。

Ada	BASIC	BLISS	C
C++	COBOL	Fortran	MACRO-32[1]
MACRO–64	Pascal	PL/I	

[1]MACRO-32 は AMACRO コンパイラでコンパイルしなければなりません。

OpenVMS Integrity でデバッガを使用する場合，次の各言語で記述されたプログラムをデバッグすることができます。

Assembler (IAS)	BASIC	BLISS	C
C++	COBOL	Fortran	MACRO-32[1]
IMACRO	PASCAL		

[1]MACRO-32 は AMACRO コンパイラでコンパイルしなければならない点に注意してください。

デバッガは，サポートされている言語の構文，データ型，演算子，式，有効範囲規則，その他の構造を認識します。 SET LANGUAGE コマンドを使用すると，デバッグ・セッションの途中で， 1 つの言語から他の言語へとデバッグ・コンテキストを変更することができます。

◆シンボリック・デバッガ

デバッガはシンボリック・デバッガです。プログラムの記憶位置は，プログラムで使用するシンボル，つまり，変数名，ルーチン名，ラベルなどによって参照できます。また，必要に応じてメモリ・アドレスやマシン・レジスタを指定することもできます。

◆すべてのデータ型のサポート

デバッガは，整数型，浮動小数点型，列挙型，レコード型，配列型などのコンパイラが生成するすべてのデータ型を認識します。プログラム変数の値は，宣言されている型に従って表示されます。

◆柔軟なデータ形式

デバッガでは，さまざまなデータ形式やデータ型を入力したり，表示できます。プログラムのソース言語によって，データの入力と表示の省略時の形式が決定されます。しかし，必要に応じて他の形式も選択できます。

◆プログラム実行の開始と再開

プログラムがデバッガによって制御されているときに，GO コマンドまたは STEP コマンドを使用すると，プログラムの実行を開始したり，再開できます。GO コマンドを実行すると，プログラムは，指定したイベントが発生するまで実行されます (たとえば，PC が指定されたコード行になるか，変数が変更されるか，例外が通知されるか，プログラムが終了するなど)。STEP コマンドを使用すると，指定した数だけ命令またはソース・コード行を実行したり，プログラムが指定されたクラスの次の命令に到達するまで実行できます。

◆ブレークポイント

SET BREAK コマンドで**ブレークポイント**を設定すると，特定の記憶位置でプログラムの実行を中断してプログラムの現在の状態をチェックすることができます。記憶位置を指定する代わりに，特定の命令クラスによって実行を停止したり，ソース行の各行で実行を停止することもできます。また，例外やタスキング (マルチスレッド) イベントなどの特定のイベントに応じて実行を中断することもできます。

◆トレースポイント

SET TRACE コマンドで**トレースポイント**を設定すると，特定の記憶位置を通るプログラムの実行パスをモニタすることができます。トレースポイントが検出されると，デバッガはトレースポイントに達したことを報告してからプログラムの実行を続けます。SET BREAK コマンドと同様に，例外イベントやタスク (マルチスレッド) イベントとして命令クラスをトレースしたり，イベントをモニタしたりすることもできます。

◆ウォッチポイント

SET WATCH コマンドで**ウォッチポイント**を設定すると，特定の変数や他のメモリ記憶位置が変更されたときに必ず実行を停止させることができます。ウォッチポイントが検出されると，デバッガがその時点で実行を中断して，変数の古い値と新しい値を報告します。

◆変数とプログラム記憶位置の操作

EXAMINE コマンドを使用すると，変数やプログラム記憶位置の値をユーザが確認することができます。それらの値を変更するには DEPOSIT コマンドを使用します。変更したあとで，その影響を調べるために実行を継続することができます。その場合にプログラムを再コンパイル，再リンク，および再実行する必要はありません。

◆式の評価

EVALUATE コマンドを使用すると，ソース言語式やアドレス式の値を求めることができます。現在デバッガに設定されている言語の構文に従って，ユーザが式と演算子を指定します。

◆制御構造

別のコマンドの実行を制御するために，各コマンドに対して論理的な制御構造 (FOR,IF, REPEAT, WHILE) を使用することができます。

◆共用可能イメージのデバッグ
共用可能イメージ (直接実行はできないイメージ) をデバッグすることができます。SET IMAGE コマンドを使用すると，(/DEBUG 修飾子によりコンパイルされたりリンクされている) 共用可能イメージの中で宣言されているシンボルをアクセスすることができます。

◆マルチプロセス・デバッグ
マルチプロセス・プログラム (複数のプロセス内で実行されるプログラム) をデバッグすることができます。 SHOW PROCESS コマンドや SET PROCESS コマンドを使用すると，プロセスの情報を表示したり，個々のプロセス内のイメージの実行を制御したりすることができます。

◆タスク・デバッグ
マルチスレッド・プログラムとも呼ばれるタスキング・プログラムをデバッグすることができます。これらのプログラムは， POSIX Threads Library サービスや POSIX 1003.1b サービス，または各言語固有のタスキング・サービス (たとえば Ada のタスキング・プログラム) を使用します。 SHOW TASK コマンドや SET TASK コマンドを使用すると，タスクの情報を表示したり，個々のタスクの実行を制御したりすることができます。

◆サポートされている端末とワークステーション
すべての VT シリーズ端末と VAX ワークステーションがサポートされています。

1.1.2 便利な機能

◆オンライン・ヘルプ
オンライン・ヘルプは，デバッガ・セッションの途中でいつでも利用することができます。オンライン・ヘルプには，すべてのデバッガ・コマンドについての情報と，選ばれたトピックについての情報が含まれています。

◆ソース・コードの表示
デバッガ・セッションでは， OpenVMS デバッガでサポートされる言語で作成されたプログラム・モジュールのソース・コードを表示できます。

◆画面モード
画面モードでは，ウィンドウにいろいろな情報を表示したり，取り込んだりすることができます。このウィンドウは，画面内で移動したりサイズを変更したりできるスクロール可能なウィンドウです。自動的に更新されるディスプレイでリース，命令，レジスタをそれぞれ見ることができます。デバッガの入出力 (I/O) や診断メッセージを表示するよう選択することもできます。また，特定のコマンド・シーケンスの出力を取り込むディスプレイ・ユニットを作成することもできます。

◆保持デバッガ
保持デバッガ (kept debugger) でデバッガを実行すると，現在のデバッガ・セッションの内部から別のプログラム・イメージを実行したり，同じイメージを再実行することができ，そのためにデバッガを終了して再起動する必要はありません。プログラムを再実行する場合には，大部分のトレースポイントやウォッチポイントをはじめ，前に設定したブレークポイントを保持するのか，取り消すのかを選択できます。

◆ DECwindows Motif ユーザ・インタフェース

OpenVMS デバッガには，オプションとして DECwindows Motif for OpenVMS グラフィカル・ユーザ・インタフェース (GUI) があり，プッシュ・ボタン，プルダウン・メニュー，ポップアップ・メニューを使用して，共通のデバッガ・コマンドにアクセスできます。 GUI はオプションとして使用できるデバッガ・コマンド行インタフェースの拡張機能であり， DECwindows Motif を実行しているワークステーションで使用できます。 GUI を使用すると， DECwindows Motif 環境で関連するすべてのデバッガ・コマンドにコマンド行から完全にアクセスできます。

◆ Microsoft Windows インタフェース

OpenVMS デバッガには，オプションとしてクライアント/サーバ構成があり，ユーザの Microsoft オペレーティング・システムを実行している PC からデバッガにアクセスして，その機能を使用することができます。このデバッガは， OpenVMS (Alpha，または Integrity CPU) 上で実行されるデバッグ・サーバと， Microsoft オペレーティング・システム (Intel または Alpha CPU) 上で実行されるデバッグ・クライアント・インタフェースで運用します。

◆クライアント/サーバ構成

クライアント/サーバ構成により， DECwindows Motif ユーザ・インタフェースを使用している OpenVMS ノードから，あるいは Microsoft Windows インタフェースを使用している PC から，特定の OpenVMS ノード上でリモートに実行しているプログラムをデバッグすることができます。多くのデバッグ・オプションを可能とする同一デバッグ・サーバに対して，同時に最大 31 のデバッグ・クライアントがアクセスすることができます。

◆キーパッド・モード

デバッガの起動時には，よく使用されるいくつかのデバッガ・コマンド・シーケンスが省略時の設定により数値キーパッドのキーに割り当てられます (VT52， VT100，または LK201 のキーボードを使用している場合)。そのため，キーボードでコマンドを入力するよりも少ないキーストロークでこれらのコマンドを入力することができます。また，ユーザが独自のキー定義を作成することもできます。

◆ソースの編集

デバッガ・セッションの途中でエラーを見つけたときに EDIT コマンドを使用すると，各自のシステムで使用可能なエディタを使用することができます。使用するエディタは SET EDITOR コマンドで設定します。ランゲージ・センシティブ・エディタ (LSE) を使用すると，画面モードのソース・ディスプレイに表示されているコードのソース・ファイルの中に自動的に編集カーソルが置かれます。

◆コマンド・プロシージャ

デバッガにコマンド・プロシージャ (複数のデバッガ・コマンドが入っているファイル) を実行させることができます。これによって，デバッガ・セッションを再現したり，直前のセッションを続行したり， 1 つのデバッガ・セッションの中で同じデバッガ・コマンドを何度も入力する手間を省いたりすることができます。さらにコマンド・プロシージャへパラメータを渡すこともできます。

◆初期化ファイル
省略時のデバッグ・モード，画面ディスプレイ定義，キーパッド・キー定義，シンボル定義などを設定するコマンドの入っている初期化ファイルを作成することができます。デバッガを起動すると，これらのコマンドが自動的に実行され，各ユーザのニーズに合ったデバッグ環境が整います。

◆ログ・ファイル
デバッガ・セッション中に入力したコマンドや，それらのコマンドに対するデバッガの応答をログ・ファイルに記録することができます。ログ・ファイルを使用するとデバッグ作業の流れを追うことができます。また，以後のデバッガ・セッションにおいてログ・ファイルをコマンド・プロシージャとして使用することもできます。

◆シンボル定義
長いコマンドやアドレス式を表現するためのシンボルや，値を短縮形で表現するためのシンボルをユーザが独自に定義することができます。

1.2 デバッグのための実行イメージの準備

プログラムをデバッガの制御下に置いて，最も効果的なシンボリック・デバッグを行うには，第 1.2.1 項および第 1.2.2 項で説明しているように，最初にコンパイラおよびリンカの/DEBUG 修飾子を使用してプログラム・モジュール (コンパイル単位) をコンパイルおよびリンクしておく必要があります。

1.2.1 デバッグのためのプログラムのコンパイル

Example 1–1 では， FORMS.C と INVENTORY.C という 2 つのソース・モジュールにより構成されているデバッグのための C プログラム， FORMS.EXE のコンパイル方法を示しています。 FORMS.C はメイン・プログラム・モジュールです。

Example 1–1 /DEBUG **修飾子によるプログラムのコンパイル**

```
$ CC/DEBUG/NOOPTIMIZE INVENTORY,FORMS
```

言語によっては，/DEBUG 修飾子や/NOOPTIMIZE 修飾子をコンパイラ・コマンドの省略時設定にしているものもあることに注意してください。この例では強調のためにこれらの修飾子を使用しています。特定の言語固有のコンパイルとリンクについては，各言語とともに提供されるドキュメントを参照してください。

Example 1–1 のコンパイラ・コマンドに指定した/DEBUG 修飾子は，オブジェクト・モジュール FORMS.OBJ と INVENTORY.OBJ で， FORMS.C と INVENTORY.C に関連付けられたシンボル情報を含むようにコンパイラに要求します。このようにすると，プログラムをデバッグしているときに，変数やルーチン，他の宣言されたシンボルのシンボル名を参照できます。シンボル情報は，/DEBUG 修飾子を使用して作成されたオブジェクト・ファイルにだけ格納されます。すべてのシ

ンボル情報を含むのか，プログラムの流れをトレースすることだけが必要なのかは，ユーザが制御できます (第 5.1.1 項を参照)。

一部のコンパイラでは，オブジェクト・コードを最適化して，プログラムのサイズを小さくしたり，実行速度を向上できます。しかし，このようにすると，オブジェクト・コードは必ずしもソース・コードと対応しなくなり，その結果，デバッグが困難になります。この状況を回避するには，/NOOPTIMIZE コマンド修飾子 (またはそれに相当する機能) を使用して，プログラムをコンパイルします。最適化されていないプログラムをデバッグした後，今度は/NOOPTIMIZE 修飾子を指定せずにプログラムを再コンパイルし，テストできます。このようにすると，最適化機能を利用できます。最適化の効果については，第 14.1 節を参照してください。

1.2.2 デバッグのためのプログラムのリンク

Example 1–2 は， FORMS.EXE という C プログラムをリンクする方法を示しています。このプログラムは FORMS.C と INVENTORY.C という 2 つのソース・モジュールで構成されます。 FORMS.C はメイン・プログラム・モジュールです。どちらのソース・モジュールも/DEBUG 修飾子を使用してコンパイルされています (Example 1–1 を参照)。

Example 1–2 /DEBUG 修飾子を使用したプログラムのリンク

```
$ LINK/DEBUG FORMS,INVENTORY
```

例 1–2 では， LINK コマンドの/DEBUG 修飾子は，リンクされているオブジェクト・モジュールに含まれているすべてのシンボル情報を実行可能イメージに含むようにリンカに要求します。大部分の言語では，インクルードするすべてのオブジェクト・モジュールを LINK コマンドに指定しなければなりません。 LINK コマンドを使用してシンボル情報を制御する方法については，第 5.1.3 項を参照してください。

Alpha システムおよび Integrity システムでは，/DSF 修飾子を使用してリンクされたプログラムをデバッグできるようになりました (したがって，別のデバッグ・シンボル・ファイルが作成されます)。 LINK コマンドに/DSF 修飾子を指定すると，リンカはシンボル情報を格納するために別の.DSF ファイルを作成します。このため，これまでより柔軟なデバッグ・オプションを選択できます。このようなプログラムをデバッグするには，次のことが必要です。

- .DSF ファイルの名前は，デバッグする.EXE ファイルの名前と一致しなければならない。
- .DSF ファイルを格納するディレクトリを指すように， DBG$IMAGE_DSF_PATH を定義しなければならない。

次の例を参照してください。

```
$ CC/DEBUG/NOOPTIMIZE TESTPROGRAM
$ LINK/DSF=TESTDISK:[TESTDIR]TESTPROGRAM.DSF TESTPROGRAM
$ DEFINE DBG$IMAGE_DSF_PATH TESTDISK:[TESTDIR]
$ DEBUG/KEEP TESTPROGRAM
```

個別のシンボル・ファイルがあるプログラムのデバッグの詳細については，第 5.1.5 項を参照してください。/DSF 修飾子の使い方については，『OpenVMS Linker Utility Manual』を参照してください。

1.2.3 LINK コマンドと RUN コマンドによるデバッガ起動の制御

実行可能なイメージにシンボル情報が渡されることに加え， LINK/DEBUG コマンドを使用すると，作成されたイメージを DCL の RUN コマンドで実行する場合，イメージ・アクティベータがデバッガを起動します。(この起動方法については，第 1.6 節を参照してください。)

コマンド修飾子/DEBUG を使用してイメージのコンパイルとリンクを行った場合でも，そのイメージをデバッガの制御下に置かずに通常どおりに実行することができます。それには DCL の RUN コマンドで/NODEBUG 修飾子を使用してください。次に例を示します。

```
$ RUN/NODEBUG FORMS
```

エラーがないと思われる場合は，この方法で簡単にプログラムをチェックすることができます。デバッガが必要とするデータは実行可能なイメージ内の領域を占有しますので，プログラムが正常であると考えられるときは/DEBUG 修飾子を使用しないで，もう一度プログラムをリンクしてください。その結果，デバッグ・シンボル・テーブルにトレースバック・データだけを持ったイメージが作成され，使用するディスク領域を節約することができます。

LINK コマンドと RUN コマンドの修飾子でデバッガの起動を制御する方法を表 1–1 に要約します。 LINK コマンドの/[NO]DEBUG 修飾子と/[NO]TRACEBACK 修飾子は，デバッガの起動だけでなく，デバッグ時に提供されるシンボル情報の最大レベルにも影響しますので注意してください。

表 1–1 LINK コマンドと RUN コマンドによるデバッガ起動の制御

LINK コマンド修飾子	デバッガなしのプログラム実行	デバッガありのプログラム実行	使用可能な最大シンボル情報[2]
/DEBUG[1]	RUN/NODEBUG	RUN	Full
なしまたは /TRACEBACK または /NODEBUG[3]	RUN	RUN/DEBUG	トレースバックのみ[4]
/NOTRACEBACK	RUN	RUN/DEBUG[5]	なし
/DSF[6]	RUN	DEBUG/KEEP[7]	すべて
/DSF[6]	RUN	DEBUG/SERVER[7]	すべて

[1]OpenVMS Alpha システムでは，デバッガやヒープ・アナライザのようにシステム・サービス・インタセプション (SSI) を使用するものは，共有リンクによって起動されたシステム・サービス呼び出しイメージを受け取ることができません。そのためイメージを起動するプログラムは，イメージがリンクされているか/DEBUG を使用して実行されている場合，共有リンクを避け，プライベート・イメージのコピーを起動するようにします。ただしこの場合，デバッガやヒープ・アナライザが制御するアプリケーションの性能に影響が現れ，共有リンクによって起動されたイメージほど高速に動作しなくなります。

[2]COMPILE コマンドの修飾子と LINK コマンドの修飾子の両方でデバッグを制御しているときに使用可能なシンボル情報のレベル (第 5.1 節を参照)。

[3]LINK/TRACEBACK (または LINK/NODEBUG) は LINK コマンドの省略時設定です。

[4]トレースバック情報には，コンパイラ生成行番号，ルーチン名，モジュール (コンパイル単位) 名が含まれます。このシンボル情報は，実行時エラーが起きたときに (実行を一時停止した地点の) PC 値とアクティブな呼び出しを示すために，トレースバック条件ハンドラが使用します。デバッガの SHOW CALLS コマンドもシンボル情報を使用します (第 2.3.3 項を参照)。

[5]RUN/DEBUG コマンドでデバッガを実行することはできますが， LINK/NOTRACEBACK コマンドを実行しているとシンボリック・デバッグを行うことはできません。

[6]Alpha および Integrity のみ

[7]論理名 DBG$DSF_IMAGE_NAME は，.DSF ファイルを格納するディレクトリを指さなければなりません (第 1.2.2 項を参照)。

1.3 保持デバッガでのプログラムのデバッグ

OpenVMS デバッガは保持デバッガで実行できます。その場合には，同じプログラムを何度も再実行することができ，また，別のプログラムを実行することもできます。そのためにデバッグ・セッションを終了する必要はありません。この節では，次の操作方法について説明します。

- 保持デバッガの起動方法，および起動後にプログラムをデバッガの制御下に置く方法
- 現在のデバッガ・セッションからの同一プログラムの再実行
- 現在のデバッガ・セッションからの他のプログラムの実行
- プログラムの実行に対する割り込み，およびデバッガ・コマンドの強制終了
- デバッガ・セッションに対する割り込み，およびそのあとのデバッグ・セッションへの復帰

1.3.1 保持デバッガの起動

この項では，DCL レベル(＄)から保持デバッガを起動して，そのあとでプログラムをデバッガの制御下に置く方法について説明します。その他の起動方法については，第 1.6 節および第 1.7 節で説明します。

第 1.3.3 項で説明する再実行機能と第 1.3.4 項の実行機能を使用するには，ここで説明する方法に従って保持デバッガを起動してください。

注意

次の問題点または制限事項は保持デバッガ固有のものです。

- **前に実行したデバッガ・プロセスが完全に停止されていない場合には，デバッガを起動するときに，次のエラーが表示されることがある。**

  ```
  %DEBUG-E-INTERR, internal debugger error in
     DBGMRPC\DBG$WAIT_FOR_EVENT got an ACK
  ```

  この問題に対処するには，デバッガを終了する。その後， DCL **の** SHOW PROCESS/SUBPROCESS **コマンドを使用して，デバッガ・サブプロセスが終了したかどうか確認する。終了している場合には，** DCL **の** STOP **コマンドを使用して停止した後，デバッガを再起動する。**

- **サイズの大きい一連のプログラムを実行すると，メモリやグローバル・セクション，あるいは他のリソースがすべて使用されてしまうため，デバッガが異常終了することがある。**

  この問題を解決するには，デバッガを終了した後，デバッガ・セッションを再起動する。

保持デバッガを起動して，プログラムをデバッガの制御下に置くには，次の手順に従ってください。

1. 第 1.2 節の説明どおりにプログラムをコンパイルおよびリンクしたことを確認する。
2. 次のコマンド行を入力する。

   ```
   $ DEBUG/KEEP
   ```

   デバッガが起動するとそのバナーが表示され，ユーザ定義の初期化ファイルが実行される (第 13.2 節を参照)。表示された DBG>プロンプトは，第 2.1 節で説明する方法でデバッガ・コマンドを入力できるようになったことを示す。
3. デバッガの RUN コマンドを使用してユーザ・プログラムをデバッガの制御下に置く。RUN コマンドには，ユーザ・プログラムの実行可能なイメージをパラメータとして指定する。次に例を示す。

   ```
   DBG> RUN FORMS
   %DEBUG-I-INITIAL,Language: C, Module: FORMS
   DBG>
   ```

表示されているメッセージは，このデバッガ・セッションが C プログラム用に初期化されており，メイン・プログラム単位 (イメージ転送アドレスを持ったモジュール) の名前が FORMS であることを示しています。初期化によって言語固有のデバッガ・パラメータが設定されます。これらのパラメータは，デバッガが名前や式を解析する方法や，デバッガの出力の書式などを制御します。言語固有のパラメータについての詳しい説明は，第 4.1.9 項を参照してください。

メイン・プログラム・ユニットを起動するときや，特定のプログラムで何らかの初期化コードを起動するときに，(一時的なブレークポイントを設定することにより) デバッガはプログラムの実行を中断します。その場合，デバッガは次のメッセージを表示します。

```
%DEBUG-I-NOTATMAIN, Type GO to reach main program
```

これらのプログラム (たとえば Ada プログラム) では，一時的なブレークポイントで完全なシンボル情報を使用して，初期化コードをデバッグすることができます。詳細については，第 14.3 節を参照してください。

第 2 章で説明されているように，これでプログラムをデバッグすることができます。

◆引数が必要なプログラムの RUN **および** RERUN **コマンド・オプション**

プログラムによっては，引数が必要なものもあります。この節では，デバッガの RUN コマンドと RERUN コマンドの使用方法，および/ARGUMENTS 修飾子と /COMMAND 修飾子でデバッガ制御を行う場合に，このようなプログラムを実行する方法について説明します。

保持デバッガでデバッガを起動した後， RUN コマンドにイメージ名を入力するか，または RUN/COMMAND コマンドと DCL フォーリン・コマンドを使用して，デバッグするイメージを指定できます。 DCL フォーリン・コマンドを指定するには，RUN コマンドの/COMMAND 修飾子を使用します。

RUN コマンドと RERUN コマンドの/ARGUMENTS 修飾子には，引数リストを指定できます。

次のデバッガ・セッションの例には，複数の方法が示されています。デバッグするプログラムは *echoargs.c* であり，入力引数を端末に表示するプログラムです。

```
#include <stdio.h>

main(int argc, char *argv[])
{
  int i;

  for (i = 0; i < argc; i++)
    printf("%s\n", argv[i]);
}
```

このプログラムを，次のようにコンパイル，リンクします。

```
$ cc/debug/noopt echoargs.c
$ link/debug echoargs
```

DCL フォーリン・コマンドは，次のように定義します。

```
$ ECHO == "$ sys$disk:[]echoargs.exe"
```

保持デバッガを起動します。次の例のデバッガ・セッションでは，引数を渡す方法として 3 種類の方法を紹介しています。

- /COMMAND と/ARGUMENTS を RUN に指定
- /ARGUMENTS を RERUN に指定
- /ARGUMENTS とイメージ名を RUN に指定

◆/COMMAND と/ARGUMENTS を指定した RUN　デバッガ・セッションのこのセクションでは，デバッガの RUN コマンドに/COMMAND 修飾子と/ARGUMENTS 修飾子を指定する方法について説明します。/COMMAND 修飾子は， DCL フォーリン・コマンド *echo*を指定します。/ARGUMENTS 修飾子は引数 *fa sol la mi*を指定します。最初の GO コマンドは*echoargs.exe*の初期化コードを実行します。その後，デバッガはプログラムの起動時に一時的なブレークポイントでプログラムの実行を中断します。 2 番目の GO コマンドは*echoargs.exe* を実行します。これは引数を画面に正しく表示します。

```
$  DEBUG/KEEP
     Debugger Banner and Version Number

DBG> RUN/COMMAND="echo"/ARGUMENTS="fa sol la mi"
%DEBUG-I-NOTATMAIN,Language: C, Module: ECHOARGS
%DEBUG-I-NOTATMAIN,Type GO to reach main program
DBG> GO
break at routine ECHOARGS\main
   1602: for (i = 0; i < argc; i++)
DBG> GO
_dsa1:[jones.test]echoargs.exe;2
fa
sol
la
mi
%DEBUG-I-EXITSTATUS,is '%SYSTEM-S-NORMAL, Normal successful completion'
```

◆/ARGUMENTS を指定した RERUN　デバッガ・セッションのこのセクションでは， RERUN コマンドの/ARGUMENTS 修飾子を使用して，新しい引数*fee fi foo fum*によって同じイメージを再実行する方法を示します (/ARGUMENTS 修飾子を指定しなかった場合には，デバッガは前に使用した引数を使用してプログラムを再実行します)。

最初の GO コマンドは*echoargs.exe*の初期化コードを実行します。そのあと，デバッガはプログラムの起動時に一時的なブレークポイントでプログラムの実行を中断します。 2 番目の GO コマンドは *echoargs.exe* を実行します。これは引数を画面に正しく表示します。

```
DBG> RERUN/ARGUMENTS="fee fii foo fum"
%DEBUG-I-NOTATMAIN,Language: C, Module: ECHOARGS
%DEBUG-I-NOTATMAIN,Type GO to reach main program
DBG> GO
break at routine ECHOARGS\main
   1602: for (i = 0; i < argc; i++)
DBG> GO
_dsa1:[jones.test]echoargs.exe;2
fee
fii
foo
fum
%DEBUG-I-EXITSTATUS,is '%SYSTEM-S-NORMAL, Normal successful completion'
```

◆/ARGUMENTS とイメージ名を指定した RUN　デバッガ・セッションのこのセクションでは， RUN コマンドを使用して*echoargs* の新しいイメージを起動し，/ARGUMENTS 修飾子を使用して引数*a b c*を指定します。

最初の GO コマンドは*echoargs.exe*の初期化コードを実行します。そのあと，デバッガはプログラムの起動時に一時的なブレークポイントでプログラムの実行を中断します。 2 番目の GO コマンドは *echoargs.exe* を実行します。これは引数を画面に正しく表示します。

```
DBG> RUN/ARGUMENTS="a b c" echoargs
%DEBUG-I-NOTATMAIN,Language: C, Module: ECHOARGS
%DEBUG-I-NOTATMAIN,Type GO to reach main program
DBG> GO
break at routine ECHOARGS\main
   1602: for (i = 0; i < argc; i++)
DBG> GO
_dsa1:[jones.test]echoargs.exe;2
a
b
c
%DEBUG-I-EXITSTATUS,is '%SYSTEM-S-NORMAL, Normal successful completion'
DBG> quit
```

◆ RUN コマンドの制限事項

デバッガの RUN コマンドについては，次の制限事項に注意してください。

- RUN コマンドを使用できるのは， DCL の DEBUG/KEEP コマンドを使用してデバッガを起動した場合のみです。
- 実行中のプログラムにデバッガを接続するために RUN コマンドを使用することはできません (第 1.7 節を参照)。

- デバッガ・クライアント/サーバ・インタフェースを使用しないと，ネットワーク・リンクを通してデバッガの制御下でプログラムを実行することはできません。デバッガ・クライアント/サーバ・インタフェースの使用についての詳細は，第 9.9 節および第 11 章を参照してください。

1.3.2 プログラム実行の終了

デバッガ・セッション中にプログラムの実行が正常に終了すると，次のメッセージが表示されます。

```
%DEBUG-I-EXITSTATUS,is '%SYSTEM-S-NORMAL, Normal successful completion')
```

このときユーザには次のオプションがあります。

- 同じデバッガ・セッションからプログラムを再実行することができる (第 1.3.3 項を参照)。
- 同じデバッガ・セッションから他のプログラムを実行することができる (第 1.3.4 項を参照)。
- デバッガ・セッションを終了することができる (第 1.8 節を参照)。

1.3.3 保持デバッガからの同じプログラムの実行

最初に第 1.3.1 項の説明どおりにデバッガを起動した場合は，現在デバッガの制御下にあるプログラムを 1 つのデバッガ・セッションの途中でいつでも再実行することができます。それには RERUN コマンドを使用してください。次に例を示します。

```
DBG> RERUN
%DEBUG-I-NOTATMAIN, Language: C, Module: ECHOARGS
%DEBUG-I-NOTATMAIN, Type GO to reach main program

DBG>
```

RERUN コマンドは，デバッグしていたイメージを終了してから，そのイメージの新しいコピーをデバッガの制御下に置きます。 RUN コマンドを使用したときと同じように，実行はメイン・ソース・モジュールの先頭で一時停止されます (第 1.3.1 項を参照)。

RERUN コマンドを使用する場合，ブレークポイント，トレースポイント，静的ウォッチポイントの現在の状態 (有効であるのか，無効であるのか) を保存できます。ただし，特定の非静的ウォッチポイントの状態は，メイン・プログラム・ユニット (実行が再起動される場所) を基準にして，ウォッチされる変数のスコープに応じて保存されないことがあります。 RERUN/SAVE は省略時の設定です。すべてのブレークポイント，トレースポイント，ウォッチポイントをクリアするには， RERUN/NOSAVE を使用します。

RERUN コマンドでは，現在デバッガの制御下にあるイメージの同一バージョンが使用されます。同じデバッガ・セッションからそのプログラムの別バージョンまたは他のプログラムをデバッグするときは， RUN コマンドを使用してください。プログラムに新しい引数を付けて再実行するとき，/ARGUMENTS 修飾子を使用します (「◆ 引数が必要なプログラムの RUN および RERUN コマンド・オプション」を参照)。

1.3.4 保持デバッガからの他のプログラムの実行

最初に第 1.3.1 項の説明どおりにデバッガを起動した場合は， 1 つのデバッガ・セッションの中でいつでも他のプログラムをデバッガの制御下に置くことができます。それには RUN コマンドを使用してください。次に例を示します。

```
DBG> RUN TOTALS
%DEBUG-I-NOTATMAIN, Language: FORTRAN, Module: TOTALS
DBG>
```

デバッガはプログラムをロードし，メイン・ソース・モジュールの先頭で実行を一時停止します。

起動の条件と制約についての詳しい説明は，第 1.3.1 項を参照してください。

RUN コマンドのすべてのオプションについての詳しい説明は， RUN コマンドの説明を参照してください。

1.4 プログラムの実行に対する割り込みとデバッガ・コマンドの強制終了

デバッガ・セッションの途中でプログラムが無限ループに陥り，そのためにデバッガのプロンプトが再表示されなくなったときは Ctrl/C を押します。これでプログラムの実行に割り込みがかかり，デバッガのプロンプトに戻ります。 Ctrl/C を押してもデバッガ・セッションは終了しません。次に例を示します。

```
DBG> GO
   .
   .
   .
Ctrl/C
DBG>
```

Ctrl/C を押すことでデバッガ・コマンドの実行を強制終了することもできます。これは，たとえば，デバッガによる長いデータ・ストリームの表示中などに役立ちます。

プログラムを実行していないとき，またはデバッガが何の処理も行っていないときは， Ctrl/C を押しても何も起こりません。

ユーザのプログラムが Ctrl/C の AST (非同期システム・トラップ) サービス・ルーチンを使用できるようにしている場合，デバッガの強制終了機能を他の Ctrl キー・シーケンスに割り当てるには， SET ABORT_KEY コマンドを使用します。現在定義されている強制終了キーを示すには， SHOW ABORT_KEY コマンドを入力します。

デバッガ・セッションの途中で Ctrl/Y を押したときの効果は，プログラムの実行中に Ctrl/Y を押した場合と同じです。 DCL コマンド・インタプリタ ($プロンプト) に制御が戻ります。

1.5 デバッガ・セッションの割り込みと再開

デバッガの SPAWN コマンドや ATTACH コマンドを使用すると，デバッガのプロンプトからデバッガ・セッションに割り込みをかけたり， DCL コマンドを入力したり，デバッガのプロンプトへ復帰したりできます。次に示すように，これらのコマンドの機能は本質的に DCL の SPAWN コマンドおよび ATTACH コマンドと同じです。

- サブプロセスを作成するにはデバッガの SPAWN コマンドを使用する。
- 既存のプロセスやサブプロセスに接続するには ATTACH コマンドを使用する。

SPAWN コマンドを入力する場合， DCL コマンドをパラメータとして指定することも指定しないこともできます。 DCL コマンドを指定すると，指定されたコマンドはサブプロセスの中で実行されます。その DCL コマンドがユーティリティを起動する場合，そのユーティリティはサブプロセス内で起動されます。 DCL コマンドの終了時またはユーザがユーティリティを終了したときに，デバッガ・セッションに制御が戻ります。 DCL コマンド DIRECTORY を実行する例を次に示します。

```
DBG> SPAWN DIR [JONES.PROJECT2]*.FOR
   .
   .
   .
%DEBUG-I-RETURNED, control returned to process JONES_1
DBG>
```

Mail ユーティリティを起動する DCL コマンド MAIL を実行する例を次に示します。

```
DBG> SPAWN MAIL
MAIL> READ/NEW
   .
   .
   .
MAIL> EXIT
%DEBUG-I-RETURNED, control returned to process JONES_1
DBG>
```

パラメータを指定しないでSPAWNコマンドを入力すると，サブプロセスが作成され，そのあとでDCLコマンドを入力することができます。サブプロセスからログ・アウトするか，またはDCLのATTACHコマンドを使用して親プロセスに接続すると，デバッガ・セッションへ戻ります。次に例を示します。

```
DBG> SPAWN
$ RUN PROG2
   .
   .
   .
$ ATTACH JONES_1
Control returned to process JONES_1
DBG>
```

デバッガ・セッションと，作成されたサブプロセス(別のデバッガ・セッションの場合もある)との間を何度も移動するときは，そのサブプロセスに接続するためにデバッガのATTACHコマンドを使用します。親プロセスへ戻るには， DCLのATTACHコマンドを使用します。デバッガを終了するたびに新しいサブプロセスを作成するわけではないので，システム・リソースがより効率的に使用されます。

2つのデバッガ・セッションを同時に実行している場合は， SET PROMPTコマンドを使用して，一方のセッション用に新しいデバッガ・プロンプトを定義することができます。これはセッションを区別するのに便利です。

1.6 プログラムの実行によるデバッガの起動

DCLのRUN *filespec*コマンドを使用すれば， 1回の手順で非保持デバッガで，デバッガの制御のもとでプログラムを起動できます。

非保持デバッガでデバッガを実行している場合には，第1.3.3項と第1.3.4項で説明したデバッガのRERUN機能とRUN機能を使用できません。デバッガの制御のもとで同じプログラムを再実行したり，別のプログラムを実行するには，デバッガをいったん終了し，再起動しなければなりません。

プログラムの実行によって非保持デバッガを開始するには，次の手順に従ってください。

1. 第1.2.1項および第1.2.2項の説明どおりにプログラムをコンパイルとリンクしたことを確認する。
2. デバッガを起動するために， DCLコマンドのRUN *filespec*を入力する。

次に例を示します。

```
$ RUN FORMS

          Debugger Banner and Version Number

%DEBUG-I-NOTATMAIN, Language: C, Module: FORMS
DBG>
```

デバッガが起動すると，そのバナーが表示され，ユーザ定義の初期化ファイルが実行されます。そして，メイン・プログラムのソース言語に合わせて言語固有のパラメータが設定されるとともに，プログラムの実行がメイン・プログラムの先頭で中断され，コマンドを要求するプロンプトが出されます。

起動の条件についての詳しい説明は，第 1.2.3 項と第 1.3.1 項を参照してください。

1.7 実行中のプログラムに割り込みをかけたあとのデバッガの起動

実行中のプログラムは自由にデバッガの制御下に置くことができます。これは，プログラムが無限ループに陥っていると思われるときや，出力が誤っていることに気付いた場合のどちらにも役立ちます。

プログラムをデバッガの制御下に置くには，次の手順に従ってください。

1. 第 1.2 節の説明どおりにプログラムをコンパイルおよびリンクしたことを確認する。
2. デバッガの制御なしでプログラムを実行するために DCL コマンドのRUN/NODEBUG *program-image*を入力する。
3. 実行中のプログラムに割り込みをかけるために Ctrl/Y を押す。 DCL のコマンド・インタプリタに制御が渡される。
4. デバッガを起動するために DCL の DEBUG コマンドを入力する。

次に例を示します。

```
$ RUN/NODEBUG FORMS
   .
   .
   .
Ctrl/Y
Interrupt
$ DEBUG

          Debugger Banner and Version Number

%DEBUG-I-NOTATMAIN, Language: C, Module: FORMS
DBG>
```

デバッガが起動すると，そのバナーが表示され，ユーザ定義の初期化ファイルが実行されます。そして，実行が割り込みをかけられた地点のモジュールのソース言語に合わせて言語固有のパラメータが設定され，コマンドを要求するプロンプトが出されます。

どこで実行に割り込みがかけられたか，普通はユーザには分かりません。実行が一時停止された地点と，呼び出しスタック上のルーチン呼び出しの並びを確認するには，SHOW CALLS コマンドを入力します (SHOW CALLS コマンドについては第 2.3.3 項を参照)。

この方法で非保持デバッガを起動した場合は，第 1.3.3 項と第 1.3.4 項でそれぞれ説明したデバッガの再実行機能と実行機能を使用することはできませんのでご注意ください。デバッガの制御下で同じプログラムを再実行する，または新しいプログラムを実行するには，いったんデバッガを終了してからもう一度起動し直す必要があります。

起動の条件についての詳しい説明は，第 1.2.3 項と第 1.3.1 項を参照してください。

1.8 デバッガ・セッションの終了

通常の方法でデバッガ・セッションを終了し， DCL レベルへ戻るには， EXIT または QUIT を入力するか， Ctrl/Z を押します。次に例をあげます。

```
DBG> EXIT
$
```

QUIT コマンドは，ログ・ファイルを閉じるためにデバッガ終了ハンドラを起動し，画面とキーパッドの状態等をリストアします。

EXIT コマンドと Ctrl/Z は同じ機能を実行します。これらは QUIT コマンドと同じ機能を実行し，さらにプログラムで宣言されている終了ハンドラも実行します。

1.9 DECwindows Motif を実行しているワークステーションでのプログラムのデバッグ

使用しているワークステーションで DECwindows Motif for OpenVMS が動いている場合，省略時の設定では，デバッガは DECwindows Motif for OpenVMS ユーザ・インタフェースで起動されます。このときに DECwindows Motif for OpenVMS ユーザ・インタフェースが表示されるのは， DECwindows Motif for OpenVMS のアプリケーション全体に通用する DECW$DISPLAY 論理名で指定されたワークステーションです。

DBG$DECW$DISPLAY 論理名を使用すると，この省略時の設定を無効にして，デバッガのコマンド・インタフェースをプログラムの入出力 (I/O) とともに DECterm ウィンドウに表示することができます。

DECterm ウィンドウにデバッガのコマンド・インタフェースを表示するには，次の手順に従ってください。

1. デバッガの起動元の DECterm ウィンドウに次の定義を入力する。

```
$ DEFINE/JOB DBG$DECW$DISPLAY " "
```

二重引用符の間には 1 つまたは複数のスペース文字を指定できる。論理名はジョブ定義を使用する。プロセス定義を使用する場合は CONFINE 属性を指定してはならない。

2. 定義を入力した DECterm ウィンドウから通常の方法でデバッガを起動する (第 1.3.1 項を参照)。そのウィンドウにデバッガのコマンド・インタフェースが表示される。

次に例を示します。

```
$ DEFINE/JOB DBG$DECW$DISPLAY " "
$ DEBUG/KEEP

          Debugger Banner and Version Number

DBG>
```

これで，第 1.3.1 項で説明したように，プログラムをデバッガの制御下に置くことができます。 DBG$DECW$DISPLAY 論理名と DECW$DISPLAY 論理名についての詳しい説明は，第 9.8.3 項を参照してください。

DECwindows Motif for OpenVMS を実行しているワークステーションでは，OpenVMS デバッガのクライアント/サーバ構成を使用することもできます。詳しい説明は，第 9.9 節を参照してください。

1.10 デバッグ・クライアントを実行している PC からのプログラムのデバッグ

OpenVMS デバッガ・バージョン 7.2 およびそれ以降の機能であるクライアント/サーバ・インタフェースを使用すると，次に示すオペレーティング・システムを実行している PC デバッグ・クライアント・インタフェースから， OpenVMS Alpha 上で実行されているプログラムをデバッグすることができます。

- Microsoft Windows (Intel)
- Microsoft Windows NT バージョン 3.51 以降 (Intel または Alpha)

注意

OpenVMS Integrity システムでは，クライアント/サーバ・インタフェースは今後のリリースで利用可能になります。

OpenVMS クライアント/サーバ構成を使用すると，次のことが可能です。

- 別の OpenVMS システムから，または Windows 95 か Windows NT バージョン 3.51 以降を実行している PC から， OpenVMS デバッグ・サーバにリモート・アクセスできる。

- クライアントは同一の，または異なる OpenVMS ノード上で実行されている複数のサーバにアクセスできる。
- 教育，またはチームでのデバッグのために，複数のクライアントが同じサーバに同時に接続できる。
- 複数の複合プラットフォーム・システムに分散された多層クライアント/サーバ・アプリケーションをデバッグできる。

クライアントとサーバは，次のいずれかのトランスポート経由で， Distributed Computing Environment/Remote Procedure Calls (DCE/RPC) を使用して通信を行います。

- TCP/IP
- UDP
- DECnet

OpenVMS ノード上でサーバを起動するには，次のコマンドを入力します。

```
$ DEBUG/SERVER
```

サーバのネットワーク・バインド文字列が表示されます。 DECwindows Motif for OpenVMS，または Microsoft Windows クライアントをサーバに接続するときは，この文字列のいずれかを指定する必要があります。例を示します。

```
$ DEBUG/SERVER
%DEBUG-I-SPEAK: TCP/IP: YES, DECnet: YES, UDP: YES
%DEBUG-I-WATCH: Network Binding: ncacn_ip_tcp:16.32.16.138[1034]
%DEBUG-I-WATCH: Network Binding: ncacn_dnet_nsp:19.10[RPC224002690001]
%DEBUG-I-WATCH: Network Binding: ncadg_ip_udp:16.32.16.138[1045]
%DEBUG-I-AWAIT: Ready for client connection...
```

クライアントの「Server Connection」ダイアログ・ボックスには，ネットワーク・プロトコル (TCP/IP， DECnet，または UDP) と，対応するネットワーク・バインド文字列を入力してください (第 9.9.4 項を参照)。

注意

省略時の設定では，サーバを起動したウィンドウに，メッセージとプログラム出力が表示されます。必要に応じて，プログラム出力を別のウィンドウにリダイレクトすることができます。

デバッグ・クライアントの使用方法の詳細については，第 11 章を参照してください。

1.11 CLI なしで動作する独立プロセスのデバッグ

デバッガの DECwindows Motif for OpenVMS ユーザ・インタフェースで設計と機能強化を行うとき，デバッグするプロセスにコマンド行インタプリタ (CLI) がなければなりません。(プリント・シンビオントのように) CLI なしで動作する，独立プロセスをデバッグするときは，デバッガに文字セル (画面モード) インタフェースを使用します。

これを行うときは， DBG$INPUT， DBG$OUTPUT， DBG$ERROR を，ログインしていない端末ポートに指定します。こうすることにより，その端末の標準文字セル・インタフェースでイメージをデバッグできるようになります。

例を示します。

```
$ DEFINE/TABLE=GROUP DBG$INPUT  TTA3:
$ DEFINE/TABLE=GROUP DBG$OUTPUT TTA3:
$ DEFINE/TABLE=GROUP DBG$ERROR  TTA3:
$ START/QUEUE SYS$PRINT /PROCESSOR=dev:[dir]test_program

[Debugger starts up on logged-out terminal TTA3:]
```

1.12 デバッガのプロセス・クォータの構成

各ユーザは，プログラムで必要とされる数のプロセスの他に，デバッガの追加サブプロセスを生成できるだけの十分な PRCLM クォータを必要とします。

BYTLM， ENQLM， FILLM， PGFLQUOTA はプールされるクォータです。次のように，デバッガ・サブプロセスを考慮して，これらのクォータの値を大きくする必要があります。

- 各ユーザの ENQLM クォータは，少なくともデバッグするプロセスの数だけ大きくしなければならない。
- 各ユーザの PGFLQUOTA を大きくしなければならない。ユーザの PGFLQUOTA が不十分な場合には，デバッガを起動することができなかったり，実行中に "virtual memory exceeded"というエラーが表示されることがある。
- 各ユーザの BYTLM クォータと FILLM クォータを大きくしなければならない。デバッグする各イメージ・ファイル，対応するソース・ファイル，デバッガ入力ファイル，出力ファイル，ログ・ファイルを開くには，十分な BYTLM クォータと FILLM クォータが必要である。これらのクォータを大きくするには，SYS$SYSTEM:AUTHORIZE.EXE を実行して， SYSUAF.DAT のパラメータを調整する。

1.13 デバッガ・コマンドの要約

次の各項では，すべてのデバッガ・コマンドおよび関係する機能別の DCL コマンドの一覧を示します。デバッガ・セッションの途中でデバッガのプロンプトで HELP と入力すると，すべてのデバッガ・コマンドとその修飾子についてのオンライン・ヘルプを参照することができます (第 2.1 節を参照)。

1.13.1 デバッガ・セッションの開始と終了

デバッガを起動するとき，プログラムをデバッガの制御下に置くとき，デバッグ・セッションに割り込みをかけるとき，またはデバッガ・セッションを終了するときに使用するコマンドを次に示します。 DCL の RUN コマンドまたは DCL の DEBUG コマンドと明記されていないコマンドはすべてデバッガ・コマンドです。

$DEBUG/KEEP	(DCL) 保持デバッガを起動する。
$RUN SYS$SHARE:DEBUGSHR.EXE	(DCL) 保持デバッガを起動する。
$DEBUG/SERVER	(DCL) デバッグ・サーバを起動する。
$DEBUG/CLIENT	(DCL) デバッグ・クライアントを起動する。
$RUN SYS$SHARE:DEBUGUISHR.EXE	(DCL) デバッグ・クライアントを起動する。
RUN *filespec*	プログラムをデバッガの制御下に置く。
RERUN	現在デバッガの制御下にあるプログラムを再実行する。
$RUN *program-image*	(DCL) 指定したイメージが LINK/DEBUG でリンクされている場合，デバッガを起動するとともに，そのイメージをデバッガの制御下に置く。この方法でデバッガを起動した場合，そのあとでデバッガの RUN コマンドや RERUN コマンドを使用することはできない。 RUN コマンドに/[NO]DEBUG 修飾子を使用すると，プログラムの実行時にデバッガを起動するかどうかを制御することができる。
EXIT, Ctrl/Z	終了ハンドラをすべて実行して，デバッガ・セッションを終了する。
QUIT	プログラム内で宣言されている終了ハンドラを何も実行しないで，デバッガ・セッションを終了する。
Ctrl/C	プログラムの実行またはデバッガ・コマンドを強制終了する。デバッガ・セッションへの割り込みは行わない。
(SET,SHOW) ABORT_KEY	省略時の Ctrl/C 強制終了機能を他の Ctrl キー・シーケンスに割り当てる (SET)。現在，強制終了機能に定義されている Ctrl キー・シーケンスを表示する (SHOW)。

Ctrl/Y $DEBUG	(DCL) デバッガの制御なしで実行しているプログラムに割り込みをかけてから，デバッガを起動する。
ATTACH	端末の制御を現在のプロセスから別のプロセスへ移す。
SPAWN	サブプロセスを作成する。これによってデバッガ・セッションを終了しないで，またデバッグ・コンテキストを失わずに，DCL コマンドを実行することができる。

1.13.2 プログラム実行の制御とモニタ

次の各コマンドは，プログラム実行を制御したりモニタしたりするときに使用します。

GO	プログラム実行を開始または再開する
STEP	次の行か次の命令まで，または指定した命令までプログラムを実行する
(SET,SHOW) STEP	STEP コマンドの省略時の修飾子を設定 (SET)，または表示 (SHOW) する
(SET,SHOW,CANCEL) BREAK	ブレークポイントの設定 (SET)，表示 (SHOW)，または取り消し (CANCEL) を行う
(ACTIVATE,DEACTIVATE) BREAK	以前に設定したブレークポイントを有効 (ACTIVATE)，または無効 (DEACTIVATE) にする
(SET,SHOW,CANCEL) TRACE	トレースポイントの設定 (SET)，表示 (SHOW)，または取り消し (CANCEL) を行う
(ACTIVATE,DEACTIVATE) TRACE	以前に設定したトレースポイントを有効 (ACTIVATE)，または無効 (DEACTIVATE) にする
(SET,SHOW,CANCEL) WATCH	ウォッチポイントの設定 (SET)，表示 (SHOW)，または取り消し (CANCEL) を行う
(ACTIVATE,DEACTIVATE) WATCH	以前に設定したウォッチポイントを有効 (ACTIVATE)，または無効 (DEACTIVATE) にする
SHOW CALLS	現在アクティブなルーチン呼び出しを表示する
SHOW STACK	現在アクティブなルーチン呼び出しについての補足情報を表示する
CALL	ルーチンを呼び出す

1.13.3 データの検査と操作

次の各コマンドは，データを検査および操作するときに使用します。

EXAMINE	変数の値，またはプログラム記憶位置の内容を表示する
SET MODE [NO]OPERANDS	命令の検査時に命令オペランドのアドレスと内容を表示するかどうかを制御する

DEPOSIT	変数の値，またはプログラム記憶位置の内容を変更する
DUMP	DCL コマンド DUMP にある程度似た，メモリの内容を表示する。
EVALUATE	言語式またはアドレス式を評価する
MONITOR	デバッガの DECwindows Motif for OpenVMS ユーザ・インタフェースにだけ適用される。変数または言語式の現在の値を DECwindows Motif for OpenVMS ユーザ・インタフェースのモニタ・ビューに表示する

1.13.4 型の選択の制御と基数の制御

次の各コマンドは，型の選択と基数を制御するときに使用します。

(SET,SHOW,CANCEL) RADIX	データを入力および表示するときの基数を設定 (SET)，表示 (SHOW)，復元 (CANCEL) する
(SET,SHOW,CANCEL) TYPE	コンパイラ生成型に対応していないプログラム記憶位置の型を設定 (SET)，表示 (SHOW)，復元 (CANCEL) する
SET MODE [NO]G_FLOAT	倍精度浮動小数点定数を G 浮動小数点数として解釈するか D 浮動小数点数として解釈するかを制御する

1.13.5 シンボル検索とシンボル化の制御

次の各コマンドは，シンボル検索とシンボル化を制御するときに使用します。

SHOW SYMBOL	プログラム内のシンボルを表示する
(SET,SHOW,CANCEL) MODULE	モジュールのシンボル情報をデバッガのシンボル・テーブルへロードすることによってモジュールを設定する (SET)。設定されたモジュールの表示 (SHOW)，または取り消し (CANCEL) を行う
(SET,SHOW,CANCEL) IMAGE	データ構造をデバッガのシンボル・テーブルにロードすることによって共用可能イメージを設定する (SET)。設定されたイメージの表示 (SHOW)，または取り消し (CANCEL) を行う
SET MODE [NO]DYNAMIC	デバッガが実行に割り込みをかけるときに，モジュールと共用可能イメージを自動的に設定するかしないかを制御する
(SET,SHOW,CANCEL) SCOPE	シンボル検索の有効範囲を設定 (SET)，表示 (SHOW)，復元 (CANCEL) する
SYMBOLIZE	メモリ・アドレスをシンボリック・アドレス式に変換する
SET MODE [NO]LINE	プログラム記憶位置を行番号で表示するか，それとも*routine-name + byte offset*で表示するかを制御する
SET MODE [NO]SYMBOLIC	プログラム記憶位置をシンボルで表示するか，それともアドレス値で表示するかを制御する

1.13.6 ソース・コードの表示

次の各コマンドは，ソース・コードの表示を制御するときに使用します。

TYPE	ソース・コードの行を表示する
EXAMINE/SOURCE	アドレス式で指定した記憶位置のソース・コードを表示する
SEARCH	指定した文字列をソース・コードから検索する
(SET,SHOW) SEARCH	SEARCH コマンドの省略時の修飾子を設定 (SET)，または表示 (SHOW) する
SET STEP [NO]SOURCE	STEP コマンド実行後のソース・コードの表示，またはブレークポイント，トレースポイント，ウォッチポイントでのソース・コードの表示を有効または無効にする
(SET,SHOW) MARGINS	ソース・コードを表示するときの左右のマージンを設定 (SET)，または表示 (SHOW) する
(SET,SHOW,CANCEL) SOURCE	ソース・ディレクトリの検索リストの作成 (SET)，表示 (SHOW)，または取り消し (CANCEL) を行う

1.13.7 画面モードの使用

次の各コマンドは，画面モードと画面ディスプレイを制御するときに使用します。

SET MODE [NO]SCREEN	画面モードを有効または無効にする
DISPLAY	ディスプレイを作成または変更する
SCROLL	ディスプレイをスクロールする
EXPAND	ディスプレイを拡大または縮小する
MOVE	ディスプレイを画面の中で移動する
(SHOW,CANCEL) DISPLAY	ディスプレイを確認 (SHOW)，または削除 (CANCEL) する
(SET,SHOW,CANCEL) WINDOW	ウィンドウの定義を作成 (SET)，表示 (SHOW)，削除 (CANCEL) する
SELECT	ディスプレイ属性に対する表示を選択する
SHOW SELECT	各ディスプレイ属性に対して選択されているディスプレイを確認する
SAVE	ディスプレイの現在の内容を別のディスプレイに保存する
EXTRACT	ディスプレイまたは現在の画面の状態をファイルに保存する
(SET,SHOW) TERMINAL	ディスプレイやその他の出力を編集するときにデバッガが使用する端末画面の高さと幅を設定 (SET)，または表示 (SHOW) する
SET MODE [NO]SCROLL	出力ディスプレイを行ごとに更新するか，それともコマンドごとに一度ずつ更新するかを制御する
Ctrl/W DISPLAY/REFRESH	画面を再表示する

1.13.8 ソース・コードの編集

次の各コマンドは，デバッガ・セッションからソースの編集を制御するときに使用します。

EDIT	デバッガ・セッションの途中でエディタを起動する
(SET,SHOW) EDITOR	EDIT コマンドで起動するエディタを設定 (SET)，または表示 (SHOW) する

1.13.9 シンボルの定義

次の各コマンドは，アドレス，コマンド，値に対する各シンボルを定義または削除するときに使用します。

DEFINE	アドレス，コマンド，または値としてシンボルを定義する
DELETE	シンボルの定義を削除する
(SET,SHOW) DEFINE	DEFINE コマンドの省略時の修飾子を設定 (SET)，または表示 (SHOW) する
SHOW SYMBOL/DEFINED	DEFINE コマンドで定義された シンボルを表示する

1.13.10 キーパッド・モードの使用

次の各コマンドは，キーパッド・モードやキー定義を制御するときに使用します。

SET MODE [NO]KEYPAD	キーパッド・モードを有効または無効にする
DEFINE/KEY	キー定義を作成する
DELETE/KEY	キー定義を削除する
SET KEY	キー定義の状態を設定する
SHOW KEY	キー定義を表示する

1.13.11 コマンド・プロシージャ，ログ・ファイル，初期化ファイルの使用

次の各コマンドは，コマンド・プロシージャやログ・ファイルとともに使用します。

@ (実行プロシージャ)	コマンド・プロシージャを実行する
(SET,SHOW) ATSIGN	コマンド・プロシージャを検索するためにデバッガが使用する省略時のファイル指定を設定 (SET)，または表示 (SHOW) する
DECLARE	コマンド・プロシージャに渡すパラメータを定義する
(SET,SHOW) LOG	デバッガのログ・ファイルを指定 (SET)，または表示 (SHOW) する
SET OUTPUT [NO]LOG	デバッガ・セッションをログに記録するかどうかを制御する

SET OUTPUT [NO]SCREEN_LOG	画面モードにおいて，画面が更新されるときに画面の内容をログに記録するかどうかを制御する
SET OUTPUT [NO]VERIFY	コマンド・プロシージャを実行するときにデバッガ・コマンドを表示するかどうかを制御する
SHOW OUTPUT	SET OUTPUT コマンドで設定した現在の出力オプションを表示する

1.13.12 制御構造の使用

次の各コマンドは，条件やループによるデバッガ・コマンドの制御構造を設定するときに使用します。

FOR	変数を増分しながらコマンドのリストを実行する
IF	条件付きでコマンドのリストを実行する
REPEAT	指定した回数だけコマンドのリストを実行する
WHILE	条件が真の間だけコマンドのリストを実行する
EXITLOOP	WHILE ループ，REPEAT ループ，または FOR ループを終了する

1.13.13 マルチプロセス・プログラムのデバッグ

次の各コマンドは，マルチプロセス・プログラムをデバッグするときに使用します。これらのコマンドはマルチプロセス・プログラムに固有のコマンドです。他のカテゴリに区分したコマンドの中にも，マルチプロセス・プログラムに固有の修飾子およびパラメータを持つコマンドがあります。たとえば，SET BREAK/ACTIVATING, EXIT *process-spec*，DISPLAY/PROCESS=などです。

CONNECT	プロセスをデバッガの制御下に置く
DEFINE/PROCESS_SET	プロセス指定のリストにシンボリック名を割り当てる
SET MODE [NO]INTERRUPT	あるプロセス内で実行を一時停止しているときに，他のプロセスの中でその実行に割り込みをかけるかどうかを制御する
(SET,SHOW) PROCESS	マルチプロセス・デバッグ環境を変更する (SET)。プロセスの情報を表示する (SHOW)
WAIT	マルチプロセス・プログラムのデバッグの際に，デバッガが新たなコマンドの入力を求める前に，すべてのプロセスが停止するのを待つかどうかを制御する

1.13.14 補助的なコマンド

次のコマンドは，他の目的で使用されます。

HELP	デバッガ・コマンドと，選択されたトピックについてのオンライン・ヘルプを表示する

ANALYZE/CRASH_DUMP	システム・ダンプ・デバッガ (SDD) で解析を行うためにプロセス・ダンプをオープンする
ANALYZE/PROCESS_DUMP	システム・コード・デバッガ (SCD) で解析を行うためにプロセス・ダンプをオープンする
(DISABLE,ENABLE,SHOW) AST	プログラム内での AST (非同期システム・トラップ) の実行要求を禁止 (DISABLE)，または可能 (ENABLE) にする。実行要求が可能か，禁止されているかを表示する (SHOW)
PTHREAD	POSIX Threads デバッガにコマンドを渡す
(SET,SHOW) EVENT_FACILITY	Ada， POSIX Threads， SCAN の各イベントの現在の実行時ファシリティを設定 (SET)，または表示 (SHOW) する
(SET,SHOW) LANGUAGE	現在の言語を設定 (SET)，または表示 (SHOW) する
SET OUTPUT [NO]TERMINAL	診断メッセージ以外のデバッガ出力を表示するかどうかを制御する
SET PROMPT	デバッガのプロンプトを指定する
(SET,SHOW) TASK | THREAD	タスキング環境を変更する (SET)。タスク情報を表示する (SHOW)
SHOW EXIT_HANDLERS	プログラム内で宣言されている終了ハンドラを表示する
SHOW MODE	SET MODE コマンドで設定した現在のデバッガのモード (たとえば，画面モードやステップ・モード) を表示する
SHOW OUTPUT	SET OUTPUT コマンドで設定した現在の出力オプションを表示する

第2部
コマンド・インタフェース

第 2 部では，デバッガのコマンド・インタフェースについて説明します。

デバッガの DECwindows Motif ユーザ・インタフェースについては，第 3 部をご覧ください。

2
デバッガの起動

本章では，デバッガの基本的なコマンド・インタフェースについて説明します。

デバッガの使用法は，デバッグの対象となるプログラムの種類，探しているエラーの種類，およびユーザの個人的な方法や経験など，いくつかの要素によって決まります。本章では，ほとんどの状況にあてはまる，次のような基本的な作業について説明します。

- デバッガ・コマンドの入力とオンライン・ヘルプの表示
- TYPE コマンドによる画面モードでのソース・コードの表示
- GO， STEP， SET BREAK の各コマンドを使用したプログラムの実行制御と，SHOW CALLS， SET TRACE， SET WATCH の各コマンドを使用したプログラムの実行のモニタ
- EXAMINE， DEPOSIT， EVALUATE の各コマンドを使用したデータの検査と操作
- パス名と SET MODULE コマンドおよび SET SCOPE コマンドを使用したシンボル参照の制御

ほとんどの例は，すべてのサポート言語に十分適用できまが，一部には，言語特有のものもあります。

第 2.6 節内のデバッグ・セッション例は，エラーを見つけ修正する方法を説明します。

デバッグ・セッションの開始と終了についての詳しい説明は，第 1.3 節を参照してください。

2.1 デバッガ・コマンドの入力とオンライン・ヘルプへのアクセス

第 1.3 節の説明どおりにいったんデバッガを起動すると，デバッガ・プロンプト (DBG>) が表示されているときはいつでも，デバッガ・コマンドを入力できます。デバッガ・コマンドを入力するには，キーボード上でコマンドを入力して Return キーを押します。たとえば，次のコマンドは変数 COUNT にウォッチポイントを設定します。

```
DBG> SET WATCH COUNT
```

デバッガ・コマンドについての詳しい説明は，『デバッガ・コマンド・ディクショナリ』またはオンライン・ヘルプで参照できます。

- ヘルプ・トピックを表示するには，プロンプトに HELP と入力する。
- ヘルプ・システムの説明を表示するには， HELP HELP と入力する。
- 完全なコマンド入力形式を表示するには，HELP Command_Formatと入力する。
- 特定のコマンドについてのヘルプを表示するには，HELP *command*と入力する。たとえば，SET WATCHコマンドのヘルプを表示するには，HELP SET WATCHと入力する。
- コマンドを機能別に表示するには，HELP Command_Summaryと入力する。

次のトピックについても，オンライン・ヘルプが使用できます。

新機能
リリース・ノート
アドレス式
組み込みシンボル
DECwindows インタフェース
キーパッド定義
言語サポート
論理名
メッセージ (診断メッセージ)
(シンボリック名を修飾する) パス名
画面モード
(プログラムからデバッガを起動する)SS$_DEBUG 条件
システム管理

上記の任意のトピックに関するヘルプを表示するには，HELP *topic* と入力します。たとえば，診断メッセージに関する情報を表示するには，HELP Messagesと入力します。

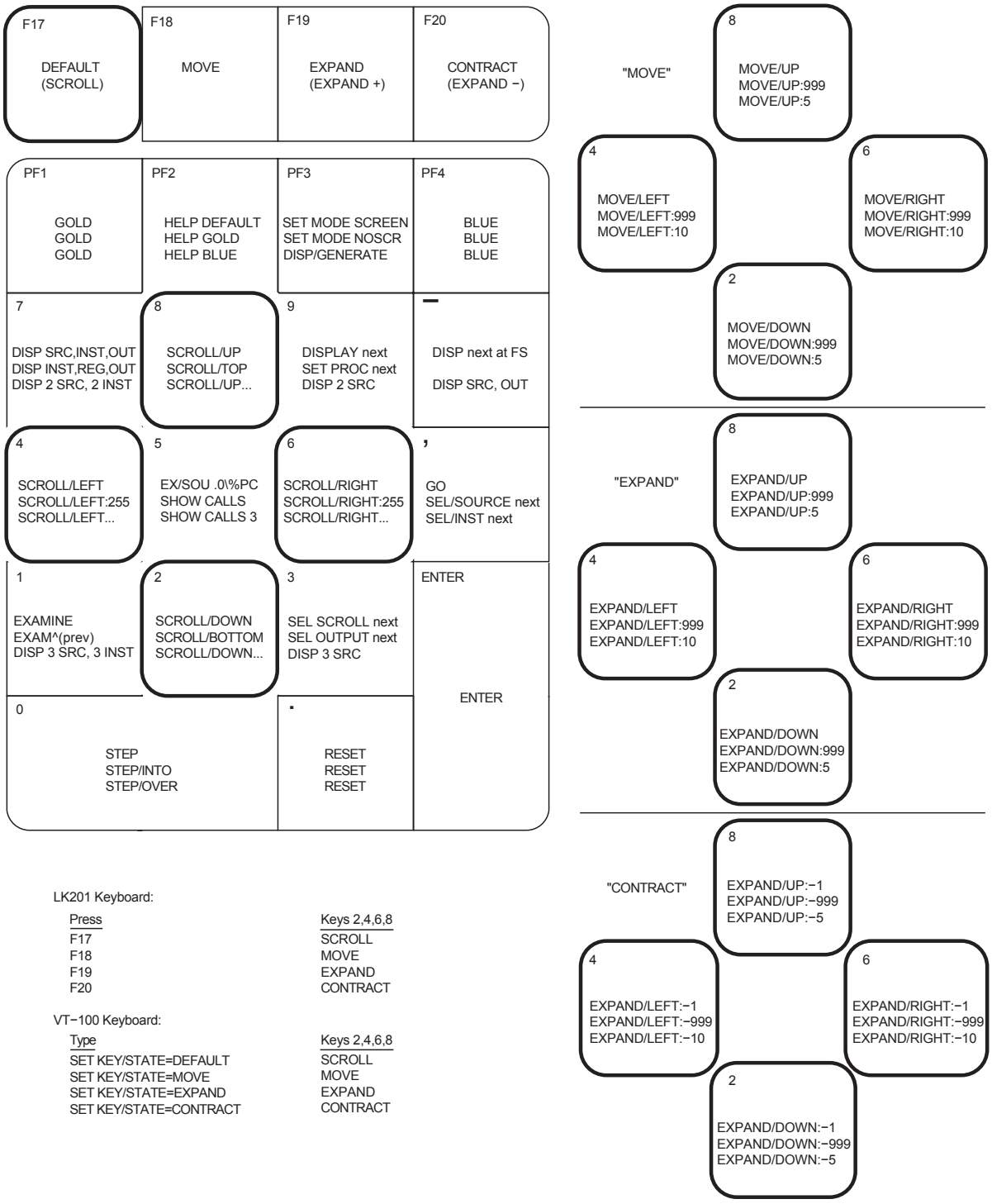
デバッガを起動すると，よく使用されるいくつかのコマンドが，自動的にメイン・キーボードの右側の数値キーパッド上のキーに割り当てられます。したがって，これらの機能は，コマンドを入力するか，またはキーパッド上のキーを押すことによって実行できます。

キーパッド上の定義済みキー機能を図 2–1 に示します。

ほとんどのキーパッド・キーには， DEFAULT， GOLD， BLUE の 3 つの定義済み機能があります。

- あるキーの DEFAULT 機能は，その目的のキーを押して実行する。

図 2–1 デバッガの定義済みキーパッド・キー機能 — コマンド・インタフェース



LK201 Keyboard:

Press	Keys 2,4,6,8
F17	SCROLL
F18	MOVE
F19	EXPAND
F20	CONTRACT

VT−100 Keyboard:

Type	Keys 2,4,6,8
SET KEY/STATE=DEFAULT	SCROLL
SET KEY/STATE=MOVE	MOVE
SET KEY/STATE=EXPAND	EXPAND
SET KEY/STATE=CONTRACT	CONTRACT

ZK−0956A−GE

- あるキーの GOLD 機能は，まず PF1 (GOLD) キーを押してから， その目的のキーを押して実行する。

- あるキーの BLUE 機能は，まず PF4 (BLUE) キーを押してから， その目的のキーを押して実行する。

図 2–1 では，各キーの中に DEFAULT， GOLD，および BLUE の各機能をそれぞれ上から下に順番に示してあります。次に例を示します。

- KP0 キー (キーパッド・キー 0) を押すと， STEP コマンドを実行できる。
- PF1 キーを押してから KP0 キーを押すと， STEP/INTO コマンドを実行できる。
- PF4 キーを押してから KP0 キーを押すと， STEP/OVER コマンドを実行できる。

通常， KP2， KP4， KP6，および KP8 の各キーは，それぞれ下方向，左方向，右方向，上方向の画面スクロールを実行します。また図 2–1 に示したように，キーボードを MOVE， EXPAND， CONTRACT の各状態に設定すると，上の 4 つのキーを使用して，画面を 4 方向に移動，拡大，または縮小できます。キーパッド・キー定義を表示するには，HELP Keypad_Definitionsコマンドを入力します。

また，キーパッド・キー機能を再定義するには， DEFINE/KEY コマンドを使用します。

2.2 ソース・コードの表示

デバッガには，非画面モードと画面モードの 2 つの情報表示モードがあります。省略時の設定では，デバッガ起動時に非画面モードになります。しかし，画面モードでソース・コードを表示したほうがよい場合があります。次の 2 つの項では， 2 つのモードについて簡単に説明します。

2.2.1 非画面モード

非画面モードは省略時の設定であり，入出力 (I/O) を行用モードで表示します。本章での会話例は，第 2.2.2 項を除き，すべて非画面モードを使用しています。

非画面モードでは， TYPE コマンドを使用して， 1 行または複数行のソース行を表示します。たとえば，次のコマンドは，現在一時停止しているモジュールの 7 行目を表示します。

```
DBG> TYPE 7
module SWAP_ROUTINES
     7:         TEMP := A;
DBG>
```

ソース行の表示は，プログラムの実行とは関係ありません。現在一時停止しているモジュール以外のモジュール (コンパイル単位) からソース・コードを表示するには，TYPE コマンドにパス名を使用して，そのモジュールを指定します。たとえば，次のコマンドは，モジュール TEST の 16 行目から 21 行目までを表示します。

```
DBG> TYPE TEST\16:21
```

パス名については， STEP コマンドとともに第 2.3.2 項で詳しく説明します。

また， EXAMINE/SOURCE コマンドを使用すれば，特定の命令と対応するルーチンまたは他の任意のプログラム記憶位置のソース行を表示することができます。

ブレークポイント，ウォッチポイント， STEP コマンドのあと，またはトレース・ポイントが検出されたとき (第 2.3 節参照) にデバッガが実行を中断する場合は，ソース行が自動的に表示されます。

プログラム内のさまざまな記憶位置のソース行を表示したあと，現在実行が一時停止している記憶位置を再表示するには， KP5 キーを押します。

表示するソース行の位置を確定できない場合，デバッガは診断メッセージを発行します。ソース行を表示できない理由は，いくつか考えられます。次に例を示します。

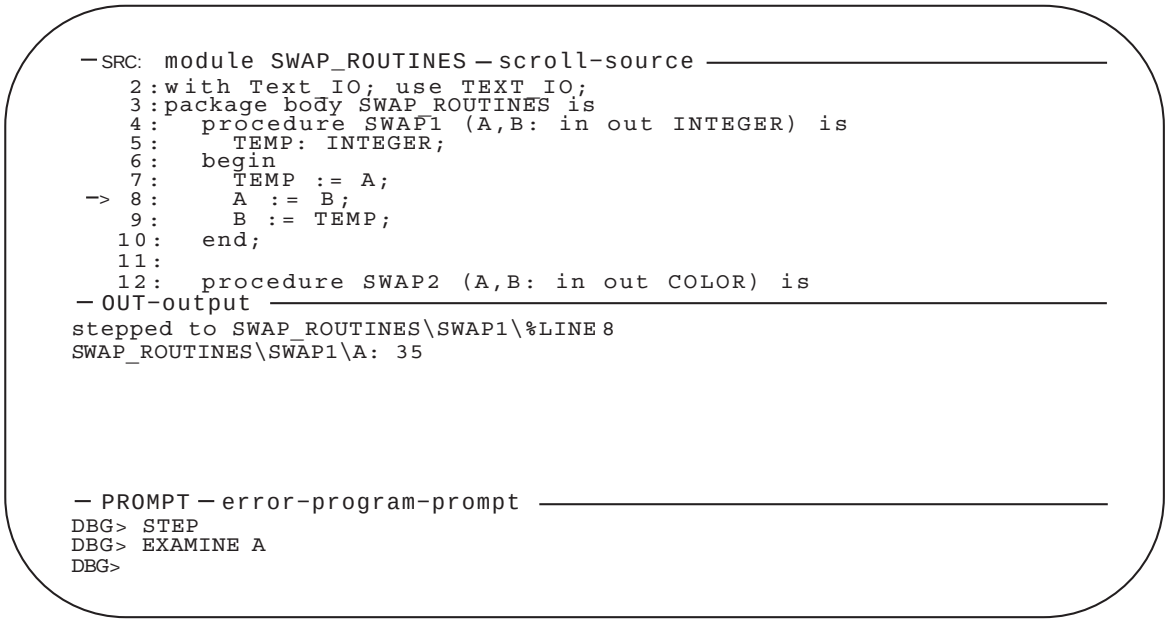
- /DEBUG 修飾子を使用しないでコンパイルまたはリンクされたモジュール内で実行が一時停止している。
- ソース・コードが使用できないシステム・ルーチンまたは共用可能イメージ・ルーチン内で実行が一時停止している。
- ソース・ファイルが，コンパイル後に別のディレクトリに移された (ソース・ファイルの記憶位置は，オブジェクト・モジュールに組み込まれている)。この場合，SET SOURCE コマンドを使用して新しい記憶位置を指定する。
- SET MODULE コマンドを使用して，モジュールを設定する必要がある。モジュールの設定については，第 2.5.1 項を参照。

画面モードから非画面モードに移行するには， PF1 キーを押したあと， PF3 キーを押すか，または SET MODE NOSCREEN を入力します。 TYPE コマンドおよび EXAMINE/SOURCE コマンドは，非画面モードだけでなく，画面モードでも使用できることに注意してください。

2.2.2 画面モード

画面モードは，ソース・コードを表示する最も簡単な方法を提供します。画面モードに切り替えるには， PF3 キーを押すか，または SET MODE SCREEN と入力します。画面モードでは，図 2–2 に示すように省略時の設定により，画面は 3 つの部分，すなわち SRC， OUT，および PROMPT に分割されます。

図 2–2 省略時の画面モード・ディスプレイ構成



ZK-6502-GE

「SRC」ディスプレイは，現在実行を一時停止しているモジュールのソース・コードを表示します。最左欄の矢印は，プログラム・カウンタ (PC) の現在値に対応するソース行を指しています。 PC は，次の実行命令のメモリ・アドレスを含むレジスタです。行番号は，コンパイラによって割り当てられますから，リスト・ファイル内の行番号と一致します。プログラムを実行するうちに，矢印は下に移動し，矢印がディスプレイの中央になるようにソース・コードは上にスクロールします。

「OUT」ディスプレイは，入力されたコマンドに対するデバッガの出力を表示します。「PROMPT」ディスプレイは，デバッガのプロンプト，ユーザの入力 (ユーザの入力したコマンド)，デバッガの診断メッセージ，およびプログラムの出力を表示します。

「SRC」ディスプレイと「OUT」ディスプレイをスクロールすれば，ディスプレイ・ウィンドウに収まらない情報もすべて，スクロールして見ることができます。スクロールするディスプレイを選択するには， KP3 キーを必要なだけ繰り返し押してください。省略時の設定では，「SRC」ディスプレイがスクロールします。上方スクロールには KP8 キーを，下方スクロールには KP2 キーを使用します。ディスプレイのスクロールは，プログラムの実行とは関係ありません。

画面モードで，実行を一時停止しているルーチンのソース行を確定できない場合，デバッガはソース行を表示できる，呼び出しスタック内の次のルーチンのソース行を表示しようとします。このようなルーチンのソース行が表示された場合，デバッガは次のメッセージを発行します。

```
%DEBUG-I-SOURCESCOPE, Source lines not available for .0\%PC.
Displaying source in a caller of the current routine.
DBG>
```

このような場合，「SRC」ディスプレイ内の矢印は，呼び出し元ルーチンの CALL 文の直後のコードを含む行を示します。

2.3 プログラム実行の制御とモニタ

この節では次の作業の実行方法について説明します。

- プログラムの実行を開始または再開する。
- プログラムを，1行単位，命令単位，または他のステップ単位で実行する。
- 現在実行が一時停止している箇所を決定する。
- ブレークポイントを使用して目的の箇所でプログラムの実行を中断する。
- トレースポイントを使用して，指定された記憶位置を通るプログラムの実行パスをトレースする。
- ウォッチポイントを使用して変数値の変化をモニタする。

これらの情報を使用すれば，第 2.4 節で説明するように，プログラム記憶位置を選択し，その記憶位置で変数の内容を検査および操作することができます。

2.3.1 プログラム実行の開始または再開

プログラムの実行を開始または再開するには， GO コマンドを使用します。

GO コマンドを使用してプログラムが開始されると，プログラムは次のいずれかのイベントが発生するまで実行を続けます。

- プログラムが実行を終了する。
- ブレークポイントに到達する。
- ウォッチポイントが検出される。
- 例外がシグナル通知される。
- ユーザが Ctrl/C を押す。

ほとんどのプログラミング言語では，プログラムがデバッガの制御下に置かれたとき，メイン・プログラムの先頭で最初に実行を一時停止します。この時点で GO コマンドを入力すると，簡単に無限ループや例外をテストできます。

実行中に無限ループが発生すると，プログラムは終了せず，デバッガのプロンプトは再表示されません。プロンプトを表示するには， Ctrl/C を押して実行に割り込みをかけます (第 1.4 節参照)。画面モードを使用している場合は，ソース・ディスプレイ

内のポインタが実行停止箇所を示しています。また， SHOW CALLS コマンドを使用すれば，呼び出しスタック内の現在アクティブなルーチン呼び出しを確認できます (第 2.3.3 項を参照)。

プログラムによって処理されない例外がシグナル通知されると，デバッガはその時点で実行を中断し，ユーザがコマンドを入力できるようにします。ユーザは，ソース・ディスプレイおよび SHOW CALLS ディスプレイを見て，実行が一時停止している箇所を見つけることができます。

GO コマンドの一般的な使用方法は，ブレークポイント，トレースポイント，およびウォッチポイントとともに使用する方法です。これらの使用方法については，それぞれ第 2.3.4 項，第 2.3.5 項，および第 2.3.6 項で説明します。実行パス内にブレークポイントを設定して GO コマンドを入力すると，そのブレークポイントで実行が一時停止します。同様に，トレースポイントを設定すると，そのトレースポイントを通して実行がモニタされます。またウォッチポイントを設定すると，ウォッチされている変数の値が変化したとき，実行が一時停止します。

2.3.2 ステップ単位でのプログラムの実行

STEP コマンドを使用すれば，プログラムを一度に 1 ステップまたはそれ以上の単位で実行できます。

省略時の設定では，ステップ単位はソース・コード 1 行です。次の例では， STEP コマンドはソース・コードを 1 行実行し，動作 ("stepped to . . . ") を報告し，次の実行行の番号 (27) とソース・コードを表示します。

```
DBG> STEP
stepped to TEST\COUNT\%LINE 27
     27:   X := X + 1;
DBG>
```

実行は，モジュール TEST 内のルーチン COUNT 内にある行 27 の最初の機械語コード命令で一時停止しています。

プログラム・シンボル (たとえば，行番号，ルーチン名，または変数名など) を表示する場合，デバッガは常に**パス名**を使用します。パス名は，シンボルとそのシンボルの記憶位置を示す接頭辞から構成されます。上の例では， TEST\COUNT\%LINE27 がパス名です。パス名の一番左の要素はモジュール名です。そのあとには，右に移動するにしたがって，そのシンボルを含むネストされたルーチンおよびブロックが続きます。バックスラッシュ(\ )は，要素を区切るために使用します。使用言語が Ada の場合は， Ada の構文にならってピリオドが使用されます。

パス名はデバッガに対して，プログラム内のシンボルを一意に識別します。通常，コマンド内でパス名を使用する必要があるのは，デバッガがプログラム内のシンボルのあいまいさを解消できないときだけです (第 2.5 節を参照)。通常デバッガは，前後関係からユーザの意味するシンボルを特定できます。

STEP コマンドを使用する場合，デバッガは，コンパイラによってコード命令が生成されたソース行だけを実行可能な行とみなします。コメント行などのその他の行は，スキップされます。

ユーザは，行単位のステップ実行の代わりに命令単位のステップ実行 (SET STEP INSTRUCTION) などの異なるステップ実行モードを指定することができます。また，省略時の設定では，呼び出されたルーチンは 1 ステップとして実行されます。すなわち，呼び出されたルーチンは実行されますが，ルーチン内で実行は一時停止されません。 SET STEP INTO コマンドを入力すれば，現在実行を一時停止しているルーチン内だけでなく，呼び出されたルーチン内でも実行を中断するようデバッガに指示できます。省略時の設定は， SET STEP OVER です。

2.3.3 実行停止箇所の決定

SHOW CALLS コマンドは，デバッグ・セッション中に実行が一時停止した箇所が分からない場合 (たとえば， Ctrl/C による割り込みのあとなど) に使用します。

このコマンドは，実行が一時停止しているルーチンに至るまでの呼び出しの並びをリストします。デバッガは，各ルーチン (実行が一時停止しているルーチンから始まる) について，次の情報を表示します。

- そのルーチンを含むモジュール名
- ルーチン名
- 呼び出しが実行された行番号 (現在のルーチンの場合は，実行が一時停止された行番号)
- 対応する PC 値。

  Alpha プロセッサと Itanium®プロセッサでは， PC 値は，モジュール内の先頭コード・アドレスからの相対アドレスと，絶対アドレスの両方の値が表示されます。

  Itanium®プロセッサには，ハードウェア PC レジスタはありません。 PC 値は，ハードウェア・インストラクション・ポインタ (IP) レジスタと，バンドル内の命令のスロット・オフセット (0， 1，または 2) を加算して，ソフトウェアにより組み立てられます。

次に例を示します。

```
DBG> SHOW CALLS
 module name   routine name   line    rel PC    abs PC

*TEST           PRODUCT        18    00000009  0000063C
*TEST           COUNT          47    00000009  00000647
*MY_PROG        MY_PROG        21    0000000D  00000653
DBG>
```

この例は，モジュール TEST 内のルーチン PRODUCT の 18 行目で実行が一時停止しており，ルーチン PRODUCT は，モジュール TEST 内のルーチン COUNT の 47 行目で呼び出され，さらにルーチン COUNT は，モジュール MY_PROG 内のルーチン MY_PROG の 21 行目で呼び出されたことを示しています。

2.3.4 ブレークポイントを使用したプログラムの実行の中断

SET BREAK コマンドを使用すれば，プログラムの実行を中断する記憶位置 (**ブレークポイント**) を選択できます。実行が中断されると，ユーザは呼び出しスタックのチェック，変数の現在値の確認などを行うことができます。 GO コマンドまたは STEP コマンドを使用して，ブレークポイントから実行を再開します。

次の例は， SET BREAK コマンドの典型的な使用方法を示したものです。

```
DBG> SET BREAK COUNT
DBG> GO
   .
   .
   .
break at routine PROG2\COUNT
     54:  procedure COUNT(X,Y:INTEGER);
DBG>
```

この例では， SET BREAK コマンドは，ルーチン COUNT (ルーチン・コードの先頭) にブレークポイントを設定しています。そして， GO コマンドで実行を開始します。ルーチン COUNT に到達すると，次のようになります。

- 実行は一時停止されます。
- デバッガは，ルーチン COUNT でブレークポイントに到達したことをユーザに知らせます ("break at . . . ")。
- デバッガは，実行を一時停止したソース行 (54) を表示します。
- デバッガは，コマンド入力を要求するプロンプトを表示します。

ユーザは，このブレークポイントで STEP コマンドを使用してルーチン COUNT をステップ実行したあと， EXAMINE コマンド (第 2.4.1 項を参照) を使用して， X および Y の値をチェックできます。

SET BREAK コマンドの使用時には，さまざまな**アドレス式** (たとえば，行番号，ルーチン名，メモリ・アドレス，バイト・オフセットなど) を使用してプログラム記憶位置を指定できます。高級言語では通常，ルーチン名，ラベル，または行番号を，必要に応じて一意性を保証するパス名とともに使用します。

ルーチン名とラベルは，ソース・コードに表示されるとおりに指定します。行番号は，ソース・コードのディスプレイまたはリスト・ファイル内から求めます。行番号を指定するには，接頭辞%LINE を使用します。この接頭辞を指定しないと，デバッガは行番号をメモリ記憶位置と解釈してしまいます。たとえば，次のコマンドは，実

行が一時停止しているモジュール内の 41 行目にブレークポイントを設定します。この結果， 41 行目の先頭で実行が中断されます。

```
DBG> SET BREAK %LINE 41
```

ブレークポイントを設定できるのは，機械語コード命令に変換された行だけであることに注意してください。それ以外の行 (たとえば，コメント行) にブレークポイントを設定しようとすると，デバッガは警告を発行します。実行が一時停止しているモジュール以外のモジュールの行番号を指定する場合は，パス名にモジュール名を含める必要があります。次に例を示します。

```
DBG> SET BREAK SCREEN_IO\%LINE 58
```

また，パラメータなしの修飾子付き SET BREAK コマンドを使用して，行ごと，または CALL 命令ごとにブレークさせることもできます。次に例を示します。

```
DBG> SET BREAK/LINE
DBG> SET BREAK/CALL
```

例外，またはタスキング・プログラム内の状態移行などの**イベント**発生時に，ブレークポイントを設定することもできます。

WHEN 句を使用してブレークポイントを条件付きにしたり， DO 句を使用してブレークポイントで実行するコマンドの並びを指定することもできます。

現在のブレークポイントを表示するには， SHOW BREAK コマンドを入力します。

ブレークポイントを無効にするには， DEACTIVATE BREAK コマンドを使用します。その場合，ブレークポイントの設定時と全く同じようにプログラム記憶位置を指定します。これで，プログラム実行中にそのブレークポイントは無効になります。しかし，あとで (たとえばそのプログラムに復帰したときに (第 1.3.3 項を参照)) 再度ブレークポイントを有効にできます。無効にされたブレークポイントは， SHOW BREAK の表示では無効であることが示されます。

ブレークポイントを有効にするには， ACTIVATE BREAK コマンドを使用します。このコマンドを使用すれば，プログラムの実行中にそのブレークポイントが有効になります。

DEACTIVATE BREAK/ALLコマンドとACTIVATE BREAK/ALLコマンドは，すべてのブレークポイントに対して作用し，特にプログラムの再実行時に有効です。

ブレークポイントを取り消すには， CANCEL BREAK コマンドを使用します。取り消されたブレークポイントは， SHOW BREAK の表示にはリストされなくなります。

2.3.5 トレースポイントを使用したプログラム実行のトレース

SET TRACE コマンドを使用すれば，プログラムの実行をトレースする記憶位置 (**トレースポイント**) を選択できます。ただし，トレースポイントでプログラムの実行が停止することはありません。トレースポイントの設定後，ユーザは GO コマンドで実行を開始し，予期しない動作をチェックしながら実行パスをモニタできます。ルーチンにトレースポイントを設定すると，そのルーチンの呼び出し回数をモニタすることもできます。

ブレークポイントと同様，トレースポイントに到達するたびに，デバッガはメッセージを発行し，ソース行を表示します。しかし，プログラムはそのまま実行を続けるため，デバッガのプロンプトは表示されません。次に例を示します。

```
DBG> SET TRACE COUNT
DBG> GO
trace at routine PROG2\COUNT
     54:  procedure COUNT(X,Y:INTEGER);
   .
   .
   .
```

ブレークポイントとトレースポイントの相違点は，この 1 点だけです。 SET TRACE コマンドの使用時には， SET BREAK コマンドの場合と全く同じアドレス式，修飾子，およびオプション句を使用します。SHOW TRACE，ACTIVATE TRACE，DEACTIVATE TRACE，CANCEL TRACEの各コマンドは，トレースポイントに対してブレークポイントの対応するコマンドと同じように動作します (第 2.3.4 項を参照)。

2.3.6 ウォッチポイントを使用した変数値の変化のモニタ

SET WATCH コマンドを使用すれば，プログラムの実行中にデバッガがモニタするプログラム変数を指定できます。このプロセスを，ウォッチポイントの設定といいます。ウォッチされている変数の値をプログラムが変更すると，デバッガは実行を中断し，情報を表示します。デバッガは，プログラムの実行中は常にウォッチポイントをモニタします。 SET WATCH コマンドは，変数だけでなく，プログラム内の任意の記憶位置をモニタするのにも使用できることに注意してください。

SET WATCH コマンドとともに変数名を指定することによって，変数にウォッチポイントを設定できます。たとえば，次のコマンドは，変数 TOTAL にウォッチポイントを設定します。

```
DBG> SET WATCH TOTAL
```

このあと，プログラムが TOTAL の値を変更するたびに，ウォッチポイントが検出されます。

注意

ウォッチポイントを設定する方法は，システム (Alpha，または Integrity)，**および変数の型，すなわち静的か非静的かにより異なります。たとえば，** Alpha **システムでは，静的変数は，プログラムの実行中，常に同じメモリ・アドレスに関連付けられています。**

次の例は，ウォッチされている変数の内容をプログラムが変更すると次に何が起こるかを示しています。

```
DBG> SET WATCH TOTAL
DBG> GO
   .
   .
   .
watch of SCREEN_IO\TOTAL at SCREEN_IO\%LINE 13
     13:   TOTAL = TOTAL + 1;
    old value: 16
    new value: 17
break at SCREEN_IO\%LINE 14
     14:   POP(TOTAL);
DBG>
```

この例では，ウォッチポイントが変数 TOTAL に設定され，実行が開始されます。変数 TOTAL の値が変化すると，実行は一時停止します。デバッガは，イベントの発生をユーザに知らせ ("watch of . . . "), TOTAL が変化した位置 (13 行目の先頭) と，対応するソース行を表示します。そし古い値と新しい値を表示し，次の行 (14 行目) の先頭で実行を一時停止します。最後に，コマンド入力を要求するプロンプトを表示します。ソース行の先頭以外で変数値が変化した場合，デバッガは行番号と行の先頭からのバイト・オフセットを表示します。

Alpha プロセッサでは，非静的変数を定義しているルーチンにトレースポイントを設定し，そのトレースポイントに到達するたびにウォッチポイントを設定する DO 句を指定することによって，その非静的変数にウォッチポイントを設定できます。**非静的変数**は，スタック上またはレジスタ内に割り当てられ，その変数を定義しているルーチンが呼び出しスタック上でアクティブな場合にしか存在しないため，常に変数名が意味を持つわけではありません (静的変数名は常に意味があります)。

次の例では，ルーチン ROUT3 内の非静的変数 Y にウォッチポイントが設定されています。トレースポイントが検出されると， WPTTRACE メッセージは非静的なウォッチポイントが設定されていることを示し， Y の値が変化するとウォッチポイントが検出されます。

```
DBG> SET TRACE/NOSOURCE ROUT3 DO (SET WATCH Y)
DBG> GO
   .
   .
   .
trace at routine MOD4\ROUT3
%DEBUG-I-WPTTRACE, nonstatic watchpoint, tracing every
                   instruction
   .
   .
   .
watch of MOD4\ROUT3\Y at MOD4\ROUT3\%LINE 16
   16:     Y := 4
   old value:    3
   new value:    4
break at MOD4\ROUT3\%LINE 17
   17:     SWAP(X,Y);
DBG>
```

呼び出し元ルーチンに制御が戻ると，非静的変数はアクティブでなくなるので，デバッガは自動的にウォッチポイントを取り消し，それを示すメッセージを発行します。

Alpha プロセッサおよび Integrity では，デバッガはすべてのウォッチポイントを非静的なウォッチポイントとみなします。

SHOW WATCH，ACTIVATE WATCH，DEACTIVATE WATCH，CANCEL WATCHの各コマンドは，ウォッチポイントに対してブレークポイントの対応するコマンドと同じように動作します (第 2.3.4 項を参照)。ただし，非静的なウォッチポイントは，実行が，ウォッチされている変数の有効範囲内に含まれる間だけ存在します。

2.4 プログラム・データの検査と操作

この節では， EXAMINE， DEPOSIT， EVALUATE の各コマンドを使用して，変数の内容の表示や変更を行ったり，式を評価したりする方法を説明します。非静的変数の値を検査したり，値を格納したりするには，第 2.3.6 項で定義したように，その非静的変数の定義ルーチンがアクティブでなければなりません。

2.4.1 変数値の表示

変数の現在の値を表示するには， EXAMINE コマンドを使用します。 EXAMINE コマンドの構文は，次のとおりです。

EXAMINE *address-expression*

デバッガは，変数のコンパイラ生成データ型を認識し，それに従ってデータを検索および編集します。次の例は， EXAMINE コマンドのいくつかの使用例です。

文字列変数の検査

```
DBG> EXAMINE EMPLOYEE_NAME
PAYROLL\EMPLOYEE_NAME:    "Peter C. Lombardi"
DBG>
```

3 つの整数変数の検査

```
DBG> EXAMINE WIDTH, LENGTH, AREA
SIZE\WIDTH:   4
SIZE\LENGTH:  7
SIZE\AREA:   28
DBG>
```

2 次元の実数配列の検査 (1 次元当たり 3 要素)

```
DBG> EXAMINE REAL_ARRAY
PROG2\REAL_ARRAY
   (1,1):       27.01000
   (1,2):       31.00000
   (1,3):       12.48000
   (2,1):       15.08000
   (2,2):       22.30000
   (2,3):       18.73000
DBG>
```

1 次元文字配列の 4 番目の要素の検査

```
DBG> EXAMINE CHAR_ARRAY(4)
PROG2\CHAR_ARRAY(4): 'm'
DBG>
```

レコード変数の検査 (COBOL の場合)

```
DBG> EXAMINE PART
INVENTORY\PART:
    ITEM:     "WF-1247"
    PRICE:     49.95
    IN_STOCK:  24
DBG>
```

レコードの構成要素の検査 (COBOL の場合)

```
DBG> EXAMINE IN_STOCK OF PART
INVENTORY\IN-STOCK of PART:
    IN_STOCK:  24
DBG>
```

EXAMINE コマンドでは，プログラム記憶位置の内容を表示する場合，変数名だけでなく任意の種類のアドレス式を使用できます。デバッガは，型未定義の記憶位置には省略時のデータ型を割り当てます。あるデータを他のデータ形式で解釈，表示したい場合には，型付きおよび型のない位置の省略時の設定を上書きすることができます。

2.4.2 変数への値の代入

変数に新しい値を代入するには， DEPOSIT コマンドを使用します。 DEPOSIT コマンドの構文は，次のとおりです。

```
DEPOSIT address-expression = language-expression
```

DEPOSIT コマンドは，ほとんどのプログラミング言語の代入文に似ています。

次の例では， DEPOSIT コマンドが，さまざまな変数に新しい値を代入しています。デバッガは，代入された値 (言語式の場合もある) が変数のデータ型と次元の制約に矛盾しないことをチェックします。

文字列の代入 (文字列は，二重引用符 (") または一重引用符 (') で囲まなければならない)

```
DBG> DEPOSIT PART_NUMBER = "WG-7619.3-84"
```

整数式の代入

```
DBG> DEPOSIT WIDTH = CURRENT_WIDTH + 10
```

文字配列の 12 番目の要素の代入 (1 つの DEPOSIT コマンドで代入できるのは配列の 1 要素だけであり，配列全体を示す集合体は代入できない)

```
DBG> DEPOSIT C_ARRAY(12) := 'K'
```

レコードの構成要素の代入 (1 つの DEPOSIT コマンドで代入できるのは，レコードの 1 構成要素だけであり，レコード全体を示す集合体は代入できない)

```
DBG> DEPOSIT EMPLOYEE.ZIPCODE = 02172
```

境界外の値の代入 (X は正の整数として宣言されている)

```
DBG> DEPOSIT X = -14
%DEBUG-I-IVALOUTBNDS, value assigned is out of bounds
        at or near DEPOSIT
```

EXAMINE コマンドの場合と同様， DEPOSIT コマンドでは，変数名だけでなく任意の種類のアドレス式を指定できます。あるデータを他のデータ形式で解釈したい場合は，型付きおよび型のない位置の省略時の設定を上書きすることができます。

2.4.3 言語式の評価

言語式を評価するには， EVALUATE コマンドを使用します。 EVALUATE コマンドの構文は，次のとおりです。

```
EVALUATE language-expression
```

デバッガは，現在使用している言語の演算子および式の構文を認識します。次の例では，整変数 WIDTH に値 45 を代入しています。そして， EVALUATE コマンドを使用して， WIDTH の現在値と 7 の合計を求めています。

```
DBG> DEPOSIT WIDTH := 45
DBG> EVALUATE WIDTH + 7
52
DBG>
```

次の例では，論理変数 WILLING と ABLE に，それぞれ TRUE と FALSE を代入しています。そして， EVALUATE コマンドを使用して，この 2 つの値の論理積を求めます。

```
DBG> DEPOSIT WILLING := TRUE
DBG> DEPOSIT ABLE := FALSE
DBG> EVALUATE WILLING AND ABLE
False
DBG>
```

2.5 プログラム内シンボルへのアクセス制御

プログラムに関連するシンボル (変数名，ルーチン名，ソース・コード，行番号など) への完全なアクセスを保証するには，第 1.2 節で説明したように，/DEBUG 修飾子を使用してプログラムをコンパイルおよびリンクしなければなりません。

これらの条件を満たしていれば，デバッガによるシンボルの処理方法は，ほとんどの場合，ユーザには見えません。ただし，次の 2 つの場合は，何らかの処置が必要です。

- モジュールの設定と取り消し
- シンボルのあいまいさの解消

2.5.1 モジュールの設定と取り消し

シンボル検索を容易にするために，デバッガは，シンボル情報を実行可能なイメージから実行時シンボル・テーブル (RST) にロードします。これで，シンボル情報に効率的にアクセスできるようになります。シンボルが RST 内に存在しない場合，デバッガはそのシンボルを認識しないか，または正確に解釈しません。

RST はメモリ領域を占有するため，デバッガは，プログラムの実行中に参照されると思われるシンボルを予想しながら，動的にシンボルをロードします。特定のモジュールのすべてのシンボル情報が，一度に RST テーブル内にロードされるため，このロード処理を**モジュール設定**と呼びます。

最初に，イメージ転送アドレスを含むモジュールだけが設定されます。その後，プログラムの実行が中断するたびに，中断ルーチンを含むモジュールが設定されます。この結果，ユーザは，その記憶位置で見えなければならないシンボルを参照できるようになります。

設定されていないモジュール内のシンボルを参照しようとすると，デバッガは，シンボルが RST に設定されていないことをユーザに警告します。次に例を示します。

```
DBG> EXAMINE K
%DEBUG-W-NOSYMBOL, symbol 'K' is not in symbol table
DBG>
```

このような場合は， SET MODULE コマンドを使用して，そのシンボルを含むモジュールを明示的に設定しなければなりません。次に例を示します。

```
DBG> SET MODULE MOD3
DBG> EXAMINE K
MOD3\ROUT2\K: 26
DBG>
```

SHOW MODULE コマンドは，プログラム内のモジュールをリストし，設定されているモジュールを識別します。

動的なモジュール設定は，設定されるモジュール数が増えるに従って，デバッガの処理速度を低下させます。性能の低下が問題になる場合は， CANCEL MODULE コマンドを使用して設定モジュール数を減らすか，または SET MODE NODYNAMIC コマンドを入力して動的モジュール設定を禁止します。動的モジュール設定を使用可能にするには， SET MODE DYNAMIC コマンドを使用します。

2.5.2 シンボルのあいまいさの解消

シンボルのあいまいさは，シンボル (たとえば，変数名 X) が 2 つ以上のルーチン内または他のプログラム単位内で定義されているときに起こります。

ほとんどの場合，デバッガはシンボルのあいまいさを自動的に解消します。最初に，現在設定されている言語の有効範囲および可視性規則が適用されます。さらに，デバッガは，任意のモジュール内のシンボル指定を許可しているため (ブレークポイントを設定するときなど)，呼び出しスタック上の呼び出しルーチンの順序で，シンボルのあいまいさを解消します。

シンボルのあいまいさを解消できない場合，デバッガは次のようなメッセージを発行します。

```
DBG> EXAMINE Y
%DEBUG-W-NOUNIQUE, symbol 'Y' is not unique
DBG>
```

このような場合は，そのシンボルの宣言を一意に指定するために，パス名接頭識別子を使用します。最初に， SHOW SYMBOL コマンドを使用して，そのシンボルに対応する (そのシンボルのすべての宣言に対応する) すべてのパス名のうち，現在 RST にロードされているものを確認します。そして，必要なパス名接頭識別子を使用して，シンボルを参照します。次に例を示します。

```
DBG> SHOW SYMBOL Y
data MOD7\ROUT3\BLOCK1\Y
data MOD4\ROUT2\Y
DBG> EXAMINE MOD4\ROUT2\Y
MOD4\ROUT2\Y: 12
DBG>
```

Y の特定の宣言を繰り返して参照する必要がある場合は， SET SCOPE コマンドを使用して，シンボル検索用に新しい省略時の有効範囲を設定します。新しい有効範囲を設定したあと，パス名接頭識別子を指定しないで Y を参照すると，新しい有効範囲内で見える Y の宣言が使用されます。次に例を示します。

```
DBG> SET SCOPE MOD4\ROUT2
DBG> EXAMINE Y
MOD4\ROUT2\Y: 12
DBG>
```

シンボル検索用の現在の有効範囲を表示するには， SHOW SCOPE コマンドを使用します。省略時の有効範囲を復元するには， CANCEL SCOPE コマンドを使用します。

2.6 デバッグ・セッションの例

この節では，論理エラーを含む FORTRAN プログラム (Example 2–1 を参照) を例に，デバッグ・セッションの概要を示します。この例には，本文の説明で参照されているソース行を識別できるよう，コンパイラ割り当て行番号が含まれています。

SQUARES という名前のこのプログラムは，次の機能を持っています。

1. データ・ファイルから一連の整数値を読み込んで，それらを配列 INARR 内に保在する (4 行目および 5 行目)。
2. ゼロ以外の各整数を 2 乗して，別の配列 OUTARR にコピーするループに入る (8 行目から 13 行目)。
3. 元の整数列内のゼロ以外の要素の数とその各要素を 2 乗した値をプリントする (16 行目から 21 行目)。

Example 2–1 **サンプル・プログラム** SQUARES

```
 1:       INTEGER INARR(20), OUTARR(20)
 2: C
 3: C     ---データ・ファイルから入力用の配列を読み込む
 4:       OPEN(UNIT=8, FILE='DATAFILE.DAT', STATUS='OLD')
 5:       READ(8,*) N, (INARR(I), I=1,N)
 6: C
 7: C     ---すべてのゼロ以外の要素を2乗してOUTARRに格納する
 8:       K = 0
 9:       DO 10 I = 1, N
10:       IF(INARR(I) .NE. 0) THEN
11:           OUTARR(K) = INARR(I)**2
12:       ENDIF
13:    10 CONTINUE
14: C
15: C     ---2乗された出力値をプリントし，終了する
16:       PRINT 20, K
17:    20 FORMAT(' Number of nonzero elements is',I4)
18:       DO 40 I = 1, K
19:       PRINT 30, I, OUTARR(I)
20:    30 FORMAT(' Element',I4,' has value',I6)
21:    40 CONTINUE
22:       END
```

SQUARES を実行すると，データ・ファイル内のゼロ以外の要素の数に関係なく，次のようなメッセージが出力されます。

```
$ RUN SQUARES
Number of nonzero elements is   0
```

このプログラムのエラーは， OUTARR 内の現在のインデックス値を保持している変数*K*が， 9 行目から 13 行目までのループ内で増分されていないことです。 11 行目の直前に K = K + 1 という文を挿入しなければなりません。

Example 2–2 に，デバッグ・セッションの開始方法とデバッガを使用してエラーを発見する方法を示します。例のあとに，各番号に対応する説明があります。

Example 2–2 **プログラム** SQUARES **を使用したデバッグ・セッション例**

```
$ FORTRAN/DEBUG/NOOPTIMIZE SQUARES   1
$ LINK/DEBUG SQUARES   2

$ DEBUG/KEEP   3

          Debugger Banner and Version Number

DBG> RUN SQUARES   4
%DEBUG-I-INITIAL, language is FORTRAN, module set to SQUARES$MAIN
```

(**次ページに続く**)

Example 2–2 (続き) プログラム SQUARES を使用したデバッグ・セッション例

```
DBG> STEP 4   5
stepped to SQUARES$MAIN\%LINE 9
     9:         DO 10 I = 1, N
DBG> EXAMINE N,K   6
SQUARES$MAIN\N:       9
SQUARES$MAIN\K:       0
DBG> STEP 2   7
stepped to SQUARES$MAIN\%LINE 11
    11:                 OUTARR(K) = INARR(I)**2
DBG> EXAMINE I,K   8
SQUARES$MAIN\I:       1
SQUARES$MAIN\K:       0
DBG> DEPOSIT K = 1   9
DBG> SET TRACE/SILENT %LINE 11 DO (DEPOSIT K = K + 1)   10
DBG> GO   11
Number of nonzero elements is   4
Element   1 has value    16
Element   2 has value    36
Element   3 has value     9
Element   4 has value    49
%DEBUG-I-EXITSTATUS, is 'SYSTEM-S-NORMAL, normal successful completion'
DBG> SPAWN   12
$ EDIT SQUARES.FOR   13
   .
   .
   .
10:       IF(INARR(I) .NE. 0) THEN
11:           K = K + 1
12:           OUTARR(K) = INARR(I)**2
13:       ENDIF
   .
   .
   .
$ FORTRAN/DEBUG/NOOPTIMIZE SQUARES   14
$ LINK/DEBUG SQUARES
$ LOGOUT   15
DBG> RUN SQUARES   16
%DEBUG-I-INITIAL, language is FORTRAN, module set to SQUARES$MAIN
DBG> SET BREAK %LINE 12 DO (EXAMINE I,K)   17
DBG> GO   18
SQUARES$MAIN\I:        1
SQUARES$MAIN\K:        1
DBG> GO
SQUARES$MAIN\I:        2
SQUARES$MAIN\K:        2
```

(次ページに続く)

Example 2–2 (続き) プログラム SQUARES を使用したデバッグ・セッション例

```
DBG> GO
SQUARES$MAIN\I:         4
SQUARES$MAIN\K:         3
DBG> EXIT   19
$
```

次の説明は， Example 2–2 内の番号に対応しています。 Example 2–1 は，デバッグ中のプログラムを示しています。

1 DCL FORTRAN コマンドの/DEBUG 修飾子は，プログラムのコードおよびデータに加えて， SQUARES に関連するシンボル情報をオブジェクト・モジュール SQUARES.OBJ に書き込むようコンパイラに指示します。

/NOOPTIMIZE 修飾子は， Fortran コンパイラの最適化を禁止して，実行可能なコードとプログラムのソース・コードが一致するようにしています。最適化されたコードをデバッグすると，いくつかのプログラム記憶位置の内容とソース・コードの表示内容とが一致しないことがあるため，混乱を招きます。

2 DCL LINK コマンドに/DEBUG 修飾子を指定すると，リンカは， SQUARES.OBJ 内に存在するすべてのシンボル情報を実行可能なイメージに含めます。

3 DCL コマンド DEBUG/KEEP が，デバッガを起動します。起動されると，バナーとデバッガ・プロンプト DBG>が表示されます。ユーザは，デバッガ・コマンドが入力できるようになります。

4 デバッガ・コマンド RUN SQUARES は，プログラム SQUARES をデバッガの制御下に置きます。情報メッセージは，プログラムのソース言語とメイン・プログラム単位名 (この例では，それぞれ FORTRAN と SQUARES) を表示します。

最初にメイン・プログラム単位の先頭 (この例では， SQUARES の 1 行目) で実行は一時停止します。

5 READ 文が実行され， K に 0 が代入されたあと，変数 N と K の値をテストします。

コマンド STEP 4 によって，プログラムのソース行が 4 行実行されます。 9 行目で実行は一時停止します。 STEP コマンドは，実行可能なコードに変換されなかったソース行を無視することに注意してください。また，省略時の設定では，デバッガは実行が一時停止したソース行を表示します。

6 コマンド EXAMINE N,K が， N と K の現在値を表示します。この時点では，これらは正しい値を示しています。

7 コマンド STEP 2 によって，プログラムはループに入り， INARR のゼロ以外のすべての要素の 2 乗を OUTARR 内にコピーします。

8 コマンド EXAMINE I,K が， I と K の現在値を表示します。

I の値は，予想どおり 1 です。しかし， K の値は 0 になっており，予想していた値 1 ではありません。ここでエラーが発見されました。 K は，ループ内の 11 行目で使用される直前に増分されなければなりません。

9 DEPOSIT コマンドを使用して， K に正しい値 1 を代入します。

10 ここで， SET TRACE コマンドを使用してプログラムをパッチし， K の値がループ内で自動的に増分されるようにします。このコマンドは，実行が 11 行目に到達するたびに検出されるトレースポイントを設定します。

- /SILENT 修飾子は，"trace at"メッセージを抑制する。/SILENT 修飾子を指定しないと， 11 行目が実行されるたびにこのメッセージが表示される。
- DO 句は，トレースポイントが検出されるたびにコマンド DEPOSIT K = K + 1 を発行する。

11 パッチをテストするために， GO コマンドで，現在の記憶位置から実行を開始します。

プログラムの出力は，パッチされたプログラムが正しく動作したことを示します。 EXITSTATUS メッセージは，プログラムが最後まで実行されたことを示します。

12 SPAWN コマンドでサブプロセスを作成して，デバッグ・セッションを終了することなく一時的に DCL レベルに制御を戻します。ここで，ソース・ファイルを修正し，プログラムを再度コンパイルおよびリンクします。

13 EDIT コマンドでエディタを起動し，ソース・ファイルを編集して， 10 行目の後ろに K = K + 1 を挿入します。例では，明確にするために，コンパイラ割り当て行番号が付加されています。

14 修正されたプログラムをコンパイルおよびリンクします。

15 作成されたサブプロセスを LOGOUT コマンドによって終了し，デバッガに制御を戻します。

16 デバッガ・コマンド RUN SQUARES は，修正されたプログラムをデバッガの制御下に置き，正しく動作することを確認できるようにします。

17 SET BREAK コマンドを使用して， 12 行目が実行されるたびに検出されるようにブレークポイントを設定します。 DO 句は，ブレークポイントが検出されると，自動的に I と K の値を表示します。

18 GO コマンドで実行を開始します。

最初のブレークポイントでは， K の値は 1 になっており，ここまではプログラムが正常に動作していることを示しています。 GO コマンドを実行するたびに I と K の現在値が表示されます。 2 つの GO コマンドのあと， K は予想どおり 3 になっていますが， I は 4 になっていることに注意してください。これは， INARR の要素の 1 つがゼロであるため， DO ループの繰り返しで 11 行目と 12 行目が実行されなかった (K が増分されなかった) からです。これで，プログラムが正しく動作していることが確認できます。

19 EXIT コマンドによってデバッグ・セッションを終了し，DCL レベルに制御を戻します。

3
プログラム実行の制御とモニタ

本章では，デバッグ中にプログラムの実行を制御およびモニタする方法を説明します。

- ステップ単位でプログラムを実行する。
- ブレークポイントにより実行を中断したり，トレースポイントにより実行をトレースしたりする。
- ウォッチポイントを使用して，変数およびその他のプログラム記憶位置の変化をモニタする。

次の 2 つの関連機能については，第 2 章を参照してください。

- GO コマンドを使用して，プログラムの実行を開始または再開する (第 2.3.1 項)。
- SHOW CALLS コマンドを使用して，現在実行が停止している箇所をモニタする (第 2.3.3 項)。

第 3.4 節では，デバッガがプログラムの実行を制御する方法について説明します。

本章には，すべてのプログラムに共通の情報が含まれています。詳細は以下の章を参照してください。

- マルチプロセス・プログラムについての詳しい説明は，第 15 章を参照。
- タスキング (マルチスレッド) プログラムについての詳しい説明は，第 16 章を参照。

現在のデバッグ・セッションから，プログラムを再実行したり，別のプログラムを実行したりする方法については，第 1.3.3 項および第 1.3.4 項をそれぞれ参照してください。

3.1 プログラムを実行するコマンド

プログラムの実行と直接関連しているデバッガ・コマンドは次の 4 つだけです。

GO
STEP
CALL
EXIT (プログラムに終了ハンドラがある場合)

第 2.3.1 項と第 2.3.2 項で説明したように， GO コマンドと STEP コマンドは，プログラムの実行を開始または再開するための基本的なコマンドです。 STEP コマンドについては，第 3.2 節で詳しく説明します。

デバッグ・セッション中は，ルーチンはプログラムの実行中に呼び出されて実行されます。 CALL コマンドを使用すれば，プログラムにリンクされたルーチンを自由に呼び出して実行できます。 CALL コマンドについては，第 13.7 節を参照してください。

EXIT コマンドについては，デバッグ・セッションの終了と関連して第 1.8 節で説明しました。 EXIT コマンドはプログラム内の任意の終了ハンドラを実行するので，終了ハンドラのデバッグにも有効です (13.6 節を参照)。

これら 4 つのどのコマンドを使用する場合も，次のイベントの発生によってプログラムの実行が中断または停止することに注意してください。

- プログラムが終了する。
- ブレークポイントに到達する。
- ウォッチポイントが検出される。
- 例外がシグナル通知される。
- ユーザが Ctrl/C を押す。

3.2 ステップ単位でのプログラムの実行

STEP コマンド (最もよく使用されるデバッガ・コマンド) を使用すれば，ユーザは，ステップ単位と呼ばれる小さい増分単位でプログラムを実行できます。

省略時の設定では，ステップ単位は実行可能なソース・コードの 1 行です。次の例では， STEP コマンドはソース・コードを 1 行実行し，動作を報告し ("stepped to . . . ")，次の実行行の番号 (27) とソース・コードを表示しています。

```
DBG> STEP
stepped to TEST\COUNT\%LINE 27
     27:   X := X + 1;
DBG>
```

実行は，モジュール TEST 内の行 27 の最初の機械語コード命令で一時停止しています。 27 行目は，モジュール TEST 内のルーチン COUNT 内に存在します。

STEP コマンドは，いくつかのソース行を一度に実行することもできます。パラメータとして正の整数を指定すると， STEP コマンドはその行数だけ実行します。次の例では， STEP コマンドは現在の位置から 3 行だけ実行します。

```
DBG> STEP 3
stepped to TEST\COUNT\%LINE 34
     34:   SWAP(X,Y);
DBG>
```

デバッガは，コンパイラがコード命令を生成したソース行だけを実行可能な行として認識することに注意してください。コメント行などの他の行はスキップされます。また，1行内に2つ以上の文が含まれる場合，デバッガは，その行のすべての命令を1ステップとして実行します。

省略時の設定では，デバッグ中のモジュールのソース行が使用可能な場合には，ステップ実行後にソース行が表示されます。コンパイル時およびリンク時に，/DEBUG 修飾子を使用していないコード (たとえば，共用可能イメージ・ルーチン) 内の命令をステップ実行する場合，ソース行は使用できません。ソース行が使用可能な場合は，SET STEP [NO]SOURCE コマンドおよび STEP コマンドの/[NO]SOURCE 修飾子を使用して，その表示を制御できます。ソース・コードの表示方法の概略，特にステップ実行後の表示方法についての説明は，第6章を参照してください。

3.2.1 STEP コマンドの動作の変更

ユーザは， STEP コマンドの省略時の動作を次の2つの方法で変更できます。

- STEP コマンド修飾子を指定する (たとえば， STEP/INTO など)。
- SET STEP コマンドを使用して，新しい省略時の修飾子を設定する (たとえば，SET STEP INTO など)。

次の例では，プログラム・カウンタ (PC) が CALL 文を指しているとき， STEP /INTO コマンドが，呼び出されたルーチン内の命令をステップ実行します。デバッガは，モジュール TEST 内のルーチン COUNT から呼び出されたルーチン PRODUCT のソース行を表示します。

```
DBG> STEP/INTO
stepped to routine TEST\PRODUCT
     6:    function PRODUCT(X,Y : INTEGER) return INTEGER is
DBG>
```

STEP/INTO コマンドの実行後は，後続の STEP コマンドは省略時の動作に戻ります。

これとは対照的に， SET STEP を使用すると， STEP コマンドの新しい省略時の動作が設定されます。いったん設定された動作は，別の SET STEP コマンドが入力されるまで有効です。たとえば， SET STEP INTO コマンドによって，後続の STEP コマンドは STEP/INTO コマンドのように動作します。 SET STEP LINE コマンドであれば，後続の STEP コマンドは STEP/LINE コマンドのように動作します。

SET STEP コマンド・パラメータは STEP コマンドの各修飾子に対応しています。

STEP コマンドに，現在の省略時の動作とは異なる修飾子を指定することによって，その STEP コマンドの間だけ現在の省略時の動作を上書きすることができます。STEP コマンドの現在の省略時の動作を示すには， SHOW STEP コマンドを使用します。

3.2.2 ルーチン内のステップ実行とルーチンの 1 ステップ実行

省略時の設定では， PC が CALL 文を指しているときに STEP コマンドを入力すると，デバッガは呼び出されたルーチンを 1 ステップとして実行します。ルーチンは実行されますが，ルーチン内で実行が停止することはなく， CALL 文の次の行の先頭で実行が停止します。命令単位で実行している場合は，呼び出されたルーチンの復帰命令の直後の命令で実行が停止します。

PC が CALL 文を指しているときに呼び出されたルーチン内の命令をステップ実行するには， STEP/INTO コマンドを入力します。次の例は， TEST モジュールのルーチン COUNT から呼び出されたルーチン PRODUCT 内の命令をステップ実行する方法を示しています。

```
DBG> STEP
stepped to TEST\COUNT\%LINE 18
    18:       AREA := PRODUCT(LENGTH, WIDTH);
DBG> STEP/INTO
stepped to routine TEST\PRODUCT
     6:    function PRODUCT(X,Y : INTEGER) return INTEGER is
DBG>
```

呼び出されたルーチン内の任意の位置から，呼び出し元のルーチンに復帰するには，STEP/RETURN コマンドを使用します。このコマンドによって，デバッガは実行中のルーチンの復帰命令をステップ実行します。後続の STEP コマンドによって，ルーチン呼び出しの直後の文に制御が戻ります。次に例を示します。

```
DBG> STEP/RETURN
stepped on return from TEST\PRODUCT\%LINE 11 to TEST\PRODUCT\%LINE 15+4
    15:    end PRODUCT;
DBG> STEP
stepped to TEST\COUNT\%LINE 19
    19:       LENGTH := LENGTH + 1;
DBG>
```

いくつかのルーチン内の命令をステップ実行するには， SET STEP INTO コマンドを入力して STEP コマンドの省略時の動作を STEP/OVER から STEP/INTO に変更します。

```
DBG> SET STEP INTO
```

このコマンドの入力後， PC が CALL 文を指しているときに STEP コマンドを入力すると，呼び出されたルーチン内で実行が中断するようになります。あとで，ルーチン呼び出しを 1 ステップとして実行したい場合には， SET STEP OVER コマンドを入力します。

SET STEP INTO が有効なときには， SET STEP コマンドに次のようなパラメータを指定することによって，デバッガがその中の命令をステップ実行する呼び出されるルーチンの種類を指定できます。

- [NO]SHARE— 呼び出された共用可能イメージ・ルーチン内の命令をステップ実行するかどうかを制御する。
- [NO]SYSTEM— 呼び出されたシステム・ルーチン内の命令をステップ実行するかどうかを制御する。

これらのパラメータを使用すれば，アプリケーションで定義されたルーチンの場合はその内部命令をステップ実行し，システム・ルーチンの場合は自動的に 1 ステップとして実行する，というような指定が可能になります。たとえば，次のコマンドは，呼び出されたルーチンがユーザ空間内に存在する場合だけ，その中の命令をステップ実行するようデバッガに指示します。システム空間内および共用可能イメージ内のルーチンは， 1 ステップとして実行されます。

```
DBG> SET STEP INTO,NOSYSTEM,NOSHARE
```

3.3 ブレークポイントによる実行の中断とトレースポイントによるトレース

この節では， SET BREAK コマンドと SET TRACE コマンドを使用して，プログラムの実行をそれぞれ中断およびトレースする方法について説明します。この 2 つのコマンドは類似点が多いため，いっしょに説明します。

SET BREAK コマンドの概要

SET BREAK コマンドを使用すれば，プログラムの実行を中断するプログラム記憶位置またはイベント (ブレークポイント) を指定できます。ブレークポイントの設定後，ユーザは， GO コマンドを使用してプログラムの実行を開始または再開できます。プログラムは，指定された記憶位置に到達するか，または指定された条件が満たされるまで，実行を続けます。ブレークポイントが検出されると，デバッガは実行を中断し，ブレークポイントを示して， DBG>プロンプトを表示します。ユーザは，ここでたとえば， SHOW CALLS コマンドで現在の位置を決定する，ルーチン内の命令をステップ実行する，変数を検査または変更する，といったデバッガ・コマンドを入力できます。

SET BREAK コマンドの構文は次のとおりです。

```
SET BREAK[/qualifier[ . . . ]] [address-expression[,  . . . ]]
[WHEN (conditional-expression)]
[DO (command[;  . . . ])]
```

次の例は， SET BREAK コマンドの典型的な使用例と，ブレークポイントでのデバッガの省略時の動作の概略を示しています。

この例では， SET BREAK コマンドは，ルーチン COUNT (ルーチン・コードの先頭) にブレークポイントを設定します。そして， GO コマンドで実行を開始します。ルーチン COUNT に到達すると，実行は一時停止されます。デバッガは，ルーチン COUNT でブレークポイントに到達したことをユーザに知らせ ("break at . . . ")，実行を停止したソース行 (54) を表示し，コマンド入力を要求するプロンプトを表示します。

```
DBG> SET BREAK COUNT
DBG> GO
   .
   .
   .
break at routine PROG2\COUNT
     54:  procedure COUNT(X,Y:INTEGER);
DBG>
```

SET TRACE コマンドの概要

SET TRACE コマンドを使用すれば，プログラムの実行をトレースするプログラム記憶位置またはイベント (トレースポイント) を選択できます。ただし，トレースポイントでプログラムが一時停止することはありません。トレースポイントの設定後，ユーザは GO コマンドで実行を開始し，予期しない動作を調べながらその記憶位置をモニタできます。ルーチンにトレースポイントを設定すると，そのルーチンの呼び出し回数をモニタすることもできます。

トレースポイントでのデバッガの省略時の動作は，ブレークポイントの場合と同じです。ただし，トレースポイントではプログラムの実行は停止しません。したがって，トレースポイントに到達し，デバッガがそれを知らせるときに， DBG>プロンプトは表示されません。

SET TRACE コマンドの構文は，コマンド名以外は SET BREAK コマンドの構文と同じです。

```
SET TRACE[/qualifier[ . . . ]] [address-expression[,  . . . ]]
[WHEN (conditional-expression)]
[DO (command[;  . . . ])]
```

SET TRACE コマンドと SET BREAK コマンドは，同じ修飾子を持っています。SET TRACE コマンド使用時には， SET BREAK コマンドの場合と全く同じように，アドレス式，修飾子，およびオプションの WHEN 句や DO 句を指定します。

SET BREAK(SET TRACE) コマンドで/TEMPORARY 修飾子を使用しないかぎり，ブレークポイントおよびトレースポイントは，ユーザが次に示す動作を行うまで有効です。

- ブレークポイント，トレースポイントを無効にする，または取り消す (第 3.3.7 項)。
- RERUN/NOSAVE コマンドを使用して，プログラムを再実行する (第 1.3.3 項)。
- 新しいプログラムを実行するか (第 1.3.4 項)，またはデバッグ・セッションを終了する (第 1.8 節)。

現在設定されているすべてのブレークポイントおよびトレースポイントを示すには，SHOW BREAK (SHOW TRACE) コマンドを使用します。

ブレークポイントまたはトレースポイントを無効にする，有効にする，または取り消すには，次のコマンドを使用します (第 3.3.7 項参照)。

DEACTIVATE BREAK, DEACTIVATE TRACE
ACTIVATE BREAK, ACTIVATE TRACE
CANCEL BREAK, CANCEL TRACE

次の各項では， SET TRACE コマンドと SET BREAK コマンドを使用してプログラム記憶位置およびイベントを指定する方法について説明します。

3.3.1 個々のプログラム記憶位置へのブレークポイントまたはトレースポイントの設定

特定のプログラム記憶位置にブレークポイントまたはトレースポイントを設定するには， SET BREAK または SET TRACE コマンドをアドレス式といっしょに指定します。

基本的には，アドレス式はメモリ・アドレスまたはレジスタを指定します。デバッガはプログラムに対応するシンボルを理解します。したがって，ユーザが通常， SET BREAK または SET TRACE コマンドといっしょに使用するアドレス式は，メモリ・アドレスではなく，ルーチン名，ラベル，またはソース行番号です。デバッガは，これらのシンボルをアドレスに変換します。

3.3.1.1 シンボリック・アドレスの指定

注意

SET BREAK コマンドまたは SET TRACE コマンドをシンボリック・アドレス式といっしょに使用する場合に，モジュールの設定，有効範囲またはパス名の指定が必要になることがあります。これらの概念については，第 5 章で詳しく説明します。以下の例では，特に指定しないかぎり，すべてのモジュールが設定され，参照されるすべてのシンボルが一意に定義されるものと想定します。

次の例は，ルーチン (SWAP) にブレークポイント，ラベル (LOOP1) にトレースポイントを設定する方法を示しています。

```
DBG> SET BREAK SWAP
DBG> SET TRACE LOOP1
```

次のコマンドは，ルーチン SWAP の復帰命令上にブレークポイントを設定します。ルーチンの復帰命令上にブレークポイントを設定すると，そのルーチンがアクティブな状態の場合でも，ローカル環境を検査することができます (たとえば，ローカル変数値の取得)。

```
DBG> SET BREAK/RETURN SWAP
```

いくつかの言語，たとえば Fortran では，数値ラベルを使用します。数値ラベル上にブレークポイントまたはトレースポイントを設定するには，数字の前に組み込みシンボルである%LABEL を付ける必要があります。%LABEL を付けないと，デバッガは，数字をメモリ・アドレスと解釈してしまいます。たとえば，次のコマンドはラベル 20 にトレースポイントを設定します。

```
DBG> SET TRACE %LABEL 20
```

行番号の前に，組み込みシンボルである%LINE を付けると，ソース・コード行にブレークポイントまたはトレースポイントを設定することができます。次のコマンドは， 14 行目にブレークポイントを設定します。

```
DBG> SET BREAK %LINE 14
```

上記のブレークポイントは， 14 行目の最初の命令で実行を停止させます。ブレークポイントまたはトレースポイントを設定できるのは，コンパイラ生成命令に対応する行 (実行可能なコード行) だけです。コメント行または変数を宣言するだけで初期化しない文など命令と対応していない行の番号を指定すると，デバッガは診断メッセージを発行します。次に例を示します。

```
DBG> SET BREAK %LINE 6
%DEBUG-I-LINEINFO, no line 6, previous line is 5, next line is 8
%DEBUG-E-NOSYMBOL, symbol '%LINE 6' is not in the symbol table
DBG>
```

上記のメッセージは，コンパイラが 6 行目と 7 行目に対応する命令を生成しなかったことを示します。

いくつかの言語では， 1 行に 2 つ以上の文を書くことができます。このような場合，同一行上の文を識別するために文番号を使用できます。文番号は，行番号の後ろにピリオド(.)とその文を示す番号を付けたものです。形式は次のとおりです。

```
%LINE line-number.statement-number
```

たとえば，次のコマンドは 38 行目の 2 番目の文にトレースポイントを設定します。

```
DBG> SET TRACE %LINE 38.2
```

コマンド内で参照されているシンボルを検索するときに，デバッガは第 5.3.1 項で説明する規則を使用します。すなわち，現在実行が停止しているモジュール内を最初に検索し，その後，呼び出しスタック内のルーチンに対応する他の有効範囲を検索します。したがって，行番号など， 2 つ以上のモジュールで定義されているシンボルを指定するには，パス名を使用する必要があります。たとえば，次のコマンドは，モジュール MOD4 の 27 行目にトレースポイントを設定します。

```
DBG> SET TRACE MOD4\%LINE 27
```

デバッガ・コマンド内で行番号を指定するときには，シンボル検索規則に注意する必要があります。実行が停止しているモジュール内で行番号が定義されていない場合，その行に対応する命令が存在しないため，デバッガはシンボル検索規則に従って，行番号が定義されている他のモジュール内を検索します。

アドレス式を指定する場合，シンボリック・アドレスとバイト・オフセットを組み合わせることができます。したがって，行番号とその行の先頭から命令の先頭バイトまでのオフセットを指定することによって，特定の命令にブレークポイントまたはトレースポイントを設定できます。たとえば，次のコマンドは， 23 行目の先頭から 5 バイトの位置のアドレスにブレークポイントを設定します。

```
DBG> SET BREAK %LINE 23+5
```

3.3.1.2 メモリ内の記憶位置の指定

メモリ内の記憶位置にブレークポイントまたはトレースポイントを設定するには，現在設定されている基数による数値アドレスを指定します。ほとんどの言語では，データの入力時と表示用の省略時の基数は， 10 進数です。

Alpha プロセッサでは，例外は， BLISS， MACRO-32， MACRO-64 です。これらの省略時の基数は， 16 進数です。

Integrity では，例外は， BLISS， MACRO–32， Intel Assembler です。

たとえば，次のコマンドは， 10 進数で 2753 のアドレスまたは 16 進数で 2753 のアドレスに，ブレークポイントを設定します。

```
DBG> SET BREAK 2753
```

組み込みシンボル%BIN，%OCT，%DEC，%HEX のいずれかを使用すると，ユーザは 2753 などの個々の整数リテラルの入力時に基数を指定できます。たとえば次のコマンド行では， 2753 が 16 進数の整数として処理されなければならないことをシンボル%HEX が指定しています。

```
DBG> SET BREAK %HEX 2753
```

数字ではなく文字で始まる 16 進数を指定するときには，最初に 0 を付ける必要があることに注意してください。そうしないと，デバッガは，指定された要素をプログラム内のシンボルと解釈しようとします。

基数の指定および組み込みシンボル%BIN，%DEC，%HEX，%OCT についての詳しい説明は，第 4.1.10 項と付録 B を参照してください。

ブレークポイントまたはトレースポイントがプログラム内のシンボルに対応する数値アドレスに設定されている場合は， SHOW BREAK コマンドまたは SHOW TRACE コマンドによって，ブレークポイントをシンボリックに示すことができます。

3.3.1.3 メモリ・アドレスの取得とシンボル化

行番号，ルーチン名，ラベルなどのシンボリック・アドレス式に対応するメモリ・アドレスを決定するには EVALUATE/ADDRESS コマンドを使用します。次に例を示します。

```
DBG> EVALUATE/ADDRESS SWAP
1536
DBG> EVALUATE/ADDRESS %LINE 26
1629
DBG>
```

アドレスは，現在の基数で表示されます。基数修飾子を指定して，別の基数でアドレスを表示することもできます。次に例を示します。

```
DBG> EVALUATE/ADDRESS/HEX %LINE 26
0000065D
DBG>
```

SYMBOLIZE コマンドは， EVALUATE/ADDRESS と逆の動作をします。つまり，対応するシンボルが存在する場合に，メモリ・アドレスをパス名を含むシンボル表現に変換します。第 5 章でシンボル化の制御方法について説明します。アドレスの取得とシンボル化についての詳しい説明は，第 4.1.11 項を参照してください。

3.3.2 行または命令へのブレークポイントまたはトレースポイントの設定

次の SET BREAK または SET TRACE コマンド修飾子を使用すれば，特定のクラス内のすべてのソース行またはすべての命令でブレークまたはトレースするように設定できます。

/LINE
/BRANCH
/CALL
/INSTRUCTION

これらの修飾子を使用する場合，アドレス式を指定してはなりません。

たとえば，次のコマンドは実行中に検出されたすべてのソース行の先頭でブレークするように設定します。

```
DBG> SET BREAK/LINE
```

命令関連の修飾子は，特に命令コードのトレースに有効です。すなわち，すべての命令をトレースする場合や特定のクラスの命令をトレースする場合に有効です。次のコマンドは，検出されたすべての分岐命令 (たとえば， BEQL， BGTR など) をトレースするように設定します。

```
DBG> SET TRACE/BRANCH
```

命令コード・トレースは，プログラムの実行速度を低下させることに注意してください。

省略時の設定では，ここで説明した修飾子を使用した場合，デバッガは現在実行しているルーチン内だけでなく，すべての呼び出されたルーチン内でもブレークまたはトレースします。これは， SET BREAK/INTO または SET TRACE/INTO を指定した場合と同じです。 SET BREAK/OVER または SET TRACE/OVER を指定すれば，すべての呼び出されたルーチン内でのブレーク動作またはトレース動作を抑止できます。あるいは，/[NO]JSB，/[NO]SHARE，/[NO]SYSTEM の各修飾子を使用して，ブレーク (トレース) 動作を抑止するルーチンの種類を指定することもできます。たとえば，次のコマンドは，呼び出されたルーチンが共用可能イメージかシステム空間内に存在する場合を除き，呼び出されたルーチン内のすべての命令でブレークするように設定します。

```
DBG> SET BREAK/LINE/NOSHARE/NOSYSTEM
```

3.3.3 エミュレートされた命令へのブレークポイントの設定 (Alpha のみ)

Alpha システムで，命令がエミュレートされているときに，デバッガがプログラムの実行を中断するように設定するには， SET BREAK/SYSEMULATE コマンドを使用します。/SYSEMULATE 修飾子を使用する場合の SET BREAK コマンドの構文は次のとおりです。

```
SET BREAK/SYSEMULATE[=mask]
```

maskは省略可能な引数であり，どの命令グループがブレークポイントを起動するかを指定するためのビットがセットされたクォドワードです。現在定義されているエミュレートされた命令グループは BYTE 命令と WORD 命令だけです。この命令グループは，maskのビット 0 を 1 にセットすることにより指定します。

maskを指定しなかった場合や，mask = FFFFFFFFFFFFFFFF の場合には，オペレーティング・システムが命令をエミュレートしたときに，デバッガはプログラムの実行を停止します。

3.3.4 ブレークポイントまたはトレースポイントでのデバッガ動作制御

SET BREAK コマンドおよび SET TRACE コマンドには，ブレークポイントおよびトレースポイントでのデバッガの動作を制御するためのオプションがいくつかあります。すなわち，/AFTER，/[NO]SILENT，/[NO]SOURCE，/TEMPORARY の各コマンド修飾子とオプションの WHEN 句と DO 句です。次に，これらのオプションの使用例をいくつか示します。

次のコマンドは， 14 行目にブレークポイントを設定し， 14 行目が 5 回実行されたあとに，そのブレークポイントが有効になるよう指定します。

```
DBG> SET BREAK/AFTER:5 %LINE 14
```

次のコマンドは，実行されるすべての行でトレースポイントが検出されるよう設定します。 DO 句は，各行が実行されるたびに変数 X の値を取得します。

```
DBG> SET TRACE/LINE DO (EXAMINE X)
```

次の例は， WHEN 句と DO 句をいっしょに取り込む方法を示したものです。このコマンドは， 27 行目にブレークポイントを設定します。ブレークポイントが検出される (実行が停止する) のは， SUM の値が 100 より大きいときだけです。 27 行目が実行されるたびにブレークするわけではありません。 DO 句は，ブレークポイントが検出されるたびに，すなわち SUM の値が 100 より大きいときは必ず，TEST_RESULT の値を調べます。実行が 27 行目に到達したときに， SUM の値が 100 以下の場合は，ブレークポイントは検出されず， DO 句も実行されません。

```
DBG> SET BREAK %LINE 27 WHEN (SUM > 100) DO (EXAMINE TEST_RESULT)
```

式"SUM> 100"などの言語式の評価についての説明は，第 4.1.6 項と第 14.3.2.2 項を参照してください。

/SILENT 修飾子は，ブレークまたはトレースのメッセージとソース・コードの表示を抑止します。この修飾子は，トレースポイントでのデバッガ・コマンドの実行だけに SET TRACE コマンドを使用したいときなどに有効です。次の例では， SET TRACE コマンドを使用して，トレースポイントで論理変数 STATUS の値を検査しています。

```
DBG> SET TRACE/SILENT %LINE 83 DO (EXAMINE STATUS)
DBG> GO
   .
   .
   .
SCREEN_IO\CLEAR\STATUS:   OFF
   .
   .
   .
```

次の例では， SET TRACE コマンドを使用して， 12 行目が実行された回数をカウントしています。最初の DEFINE/VALUE コマンドは，シンボル COUNT を定義し，その値を 0 に初期化します。 SET TRACE コマンドの DO 句は， COUNT を増分して，トレースポイントが検出されるたびに (12 行目に到達するたびに) その値を評価します。

```
DBG> DEFINE/VALUE COUNT=0
DBG> SET TRACE/SILENT %LINE 12 DO (DEF/VAL COUNT=COUNT+1;EVAL COUNT)
```

ソース行は，省略時の設定では，デバッグ中のモジュールのソース・コードが使用可能な場合に，ブレークポイント，トレースポイント，およびウォッチポイントで表示されます。ユーザは， SET BREAK， SET TRACE， SET WATCH の各コマンドの/[NO]SOURCE 修飾子および SET STEP [NO]SOURCE コマンドを使用して，ソース行の表示を制御できます。ソース・コードの表示制御の概略，特にブレークポイント，トレースポイント，およびウォッチポイントでの表示制御についての説明は，第 6 章を参照してください。

3.3.5 例外発生時点でのブレークポイントまたはトレースポイントの設定

SET BREAK/EXCEPTION コマンドおよび SET TRACE/EXCEPTION コマンドは，プログラムによって生成された任意の例外をそれぞれブレークポイントまたはトレースポイントとして扱うようデバッガに指示します。ブレークポイントまたはトレースポイントは，アプリケーションによって定義された例外ハンドラの起動前に発生します。例外ハンドラおよび条件ハンドラに関連するデバッグ方法については， 13.5 節を参照してください。

3.3.6 イベント発生時点でのブレークポイントまたはトレースポイントの設定

SET BREAK コマンドおよび SET TRACE コマンドには，それぞれ/EVENT=*event-name*修飾子を使用できます。ユーザは，この修飾子を使用して，イベント名キーワードで示されるさまざまなイベントによって検出されるブレークポイントまたはトレースポイントを設定できます。イベントおよびそのキーワードは現在，次のイベント機能について定義されています。

- ADA イベント機能。 Ada のタスキング・イベントを定義する。 Ada は第 16.6.4 項で定義されている。
- THREADS イベント機能。 POSIX Threads サービスを使用している任意の言語で記述されたプログラムのタスキング (マルチスレッド) イベントを定義する。 THREADS イベントは第 16.6.4 項で定義されている。

適切な機能およびイベント名キーワードは，プログラムがデバッガの制御下に置かれたときに定義されます。現在のイベント機能と，対応するイベント名キーワードを示すには， SHOW EVENT_FACILITY コマンドを使用します。 SET EVENT_FACILITY コマンドを使用すれば，ユーザはイベント機能を変更でき，デバッグ・コ

ンテキストも変更できます。これは，複数言語プログラムの使用時に，あるイベント機能に対応するルーチンをデバッグしたいが，その機能が現在設定されていない，という場合に有効です。

次の例は， SCAN イベント・ブレークポイントの設定方法を示しています。このコマンドによって，デバッガは，任意の値に対して SCAN トークンが作成されると必ずブレークするようになります。

```
DBG> SET BREAK/EVENT=TOKEN
```

ブレークポイントまたはトレースポイントが検出されると，デバッガは検出される原因となったイベントを示し，補足情報を提供します。

3.3.7 ブレークポイントまたはトレースポイントの無効化，有効化，および取り消し

いったん設定されたブレークポイントまたはトレースポイントは，無効にしたり，有効にしたり，また取り消したりすることができます。

ブレークポイントまたはトレースポイントを無効にするには，DEACTIVATE BREAKまたはDEACTIVATE TRACEコマンドを入力します。これで，デバッガは，プログラムの実行中にそのブレークポイントまたはトレースポイントを無視するようになります。いったん無効にしたブレークポイントまたはトレースポイントはあとで，たとえば呼び出し元のプログラムを再実行したときなど (第 1.3.3 項参照) に再度有効にすることもできます。無効にされたブレークポイントまたはトレースポイントは，SHOW BREAKコマンドを実行したときに表示されるリストでは無効なもとのして表示されます。

ブレークポイントまたはトレースポイントを有効にするには，ACTIVATE BREAKまたはACTIVATE TRACEコマンドを使用します。これらのコマンドを使用すれば，プログラムの実行中にブレークポイントまたはトレースポイントが有効になります。

DEACTIVATE BREAK/ALLおよびACTIVATE BREAK/ALLコマンド (DEACTIVATE TRACE/ALLおよびACTIVATE TRACE/ALLコマンド) は，すべてのブレークポイントまたはトレースポイントに対して作用し，特に RERUN コマンドを使用してプログラムを再実行するときに有効です。

ブレークポイントまたはトレースポイントを取り消すには，CANCEL BREAKまたはCANCEL TRACEコマンドを使用します。取り消されたブレークポイントまたはトレースポイントは，SHOW BREAKまたはSHOW TRACEコマンドで表示されるリストには載らなくなります。

これらのコマンドを使用する場合には，ブレークポイントまたはトレースポイントの設定時と全く同じようにして，アドレス式と (必要であれば) 修飾子を指定します。次に例を示します。

```
DBG> DEACTIVATE TRACE/LINE
DBG> CANCEL BREAK SWAP,MOD2\LOOP4,2753
```

3.4 変数および他のプログラム記憶位置の変更のモニタ

SET WATCH コマンドを使用すれば，デバッガがプログラムの実行中にモニタするプログラム変数または任意のメモリ記憶位置を指定できます。このプロセスを，ウォッチポイントの設定といいます。ウォッチされている変数またはメモリ記憶位置の値がプログラム実行中に変更されると，ウォッチポイントが検出されます。デバッガは実行を中断し情報を表示して，コマンド入力用プロンプトを表示します。デバッガは，プログラムの実行中は常にウォッチポイントをモニタします。

この節では， SET WATCH コマンドの使用方法の概略を説明します。非静的変数，すなわち呼び出しスタック上またはレジスタ内に割り当てられる変数へのウォッチポイントの設定についての詳しい説明は，第 3.4.3 項を参照してください。

注意

SET WATCH **コマンドを変数名または任意のシンボリック・アドレス式といっしょに使用する場合，モジュールの設定あるいは有効範囲かパス名の指定が必要になることがあります。これらの概念については，第** 5 **章で詳しく説明します。以下の例では，すべてのモジュールが設定され，すべての変数名が一意に定義されるものと想定します。**

コンパイル時にプログラムの最適化を行うと，プログラム内のある変数がコンパイラによって除去されることがあります。このような変数にウォッチポイントを設定しようとすると，デバッガは警告を発行します (第 1.2 **節および第** 14.1 **節を参照)。**

SET WATCH コマンドの構文は次のとおりです。

```
SET WATCH[/qualifier[ . . . ]] address-expression[, . . . ]
[WHEN (conditional-expression)]
[DO (command[; . . . ])]
```

ユーザは，任意の有効なアドレス式を指定できますが，通常は変数名を指定します。次に，典型的な SET WATCH コマンドの使用例と，ウォッチポイントでのデバッガの省略時の動作の概略を示します。

```
DBG> SET WATCH COUNT
DBG> GO
   .
   .
   .
watch of MOD2\COUNT at MOD2\%LINE 24
     24:   COUNT := COUNT + 1;
    old value: 27
    new value: 28
break at MOD2\%LINE 25
     25:   END;
DBG>
```

この例では， SET WATCH コマンドによって，変数 COUNT にウォッチポイントが設定され， GO コマンドで実行が開始されます。プログラムが変数 COUNT の値を変更すると，実行は一時停止します。その後デバッガは次の作業を実行します。

- イベントの発生を知らせ ("watch of MOD2\COUNT . . . ")，ウォッチされている変数の値を変更した命令の記憶位置を表示する (" . . . at MOD2\%LINE 24")。
- 関連するソース行 (24) を表示する。
- 変数の古い値と新しい値 (27 と 28) を表示する。
- 次の行の先頭で実行が停止していることを知らせ ("break at MOD2\%LINE 25")，そのソース行を表示する。
- 別のコマンド入力用のプロンプトを表示する。

ウォッチされている変数の値を変更した命令のアドレスが，ソース行の先頭でない場合，デバッガは，行番号に行の先頭からのバイト・オフセットを追加することによって，その命令の記憶位置を示します。次に例を示します。

```
DBG> SET WATCH K
DBG> GO
   .
   .
   .
watch of TEST\K at TEST\%LINE 19+5
     19:   DO 40 K = 1, J
    old value: 4
    new value: 5
break at TEST\%LINE 19+9
     19:   DO 40 K = 1, J
DBG>
```

この例では，変数 K を変更した命令のアドレスは， 19 行目の先頭から 5 バイトの位置です。ブレークポイントは，変数値を変更した命令の次の命令にあることに注意してください。前の例のように次のソース行の先頭ではありません。

ウォッチポイントは，集合体，すなわち全配列および全レコードに設定することができます。配列またはレコードに設定されたウォッチポイントは，その配列またはレコード内の任意の要素が変更されると検出されます。したがって，個々の配列要素またはレコード構成要素にウォッチポイントを設定する必要はありません。ただし，可変長レコードには集合体ウォッチポイントを設定できません。次の例では，配列 ARR の第 3 要素が変更されたためウォッチポイントが検出されています。

```
DBG> SET WATCH ARR
DBG> GO
   .
   .
   .
watch of SUBR\ARR at SUBR\%LINE 12
    12:     ARR(3) := 28
   old value:
    (1):         7
    (2):         12
    (3):         3
    (4):         0

   new value:
    (1):         7
    (2):         12
    (3):         28
    (4):         0

break at SUBR\%LINE 13
DBG>
```

ウォッチポイントは，レコードの構成要素，個々の配列要素，または配列断面 (配列要素の範囲) にも設定できます。配列断面に設定されたウォッチポイントは，その配列断面内の任意の要素が変更されたときに検出されます。ウォッチポイント設定時には，現在の言語の構文を使用します。たとえば，次のコマンドは， Pascal の構文を使用して配列 CHECK の第 7 要素にウォッチポイントを設定します。

```
DBG> SET WATCH CHECK[7]
```

現在設定されているすべてのウォッチポイントを示すには， SHOW WATCH コマンドを使用します。

3.4.1 ウォッチポイントの無効化，有効化，および取り消し

いったん設定されたウォッチポイントは，無効にしたり，有効にしたり，また取り消したりすることができます。

ウォッチポイントを無効にするには，DEACTIVATE WATCHコマンドを入力します。これで，デバッガは，プログラムの実行中にウォッチポイントを無視するようになります。いったん無効にしたウォッチポイントをあとで，たとえば呼び出し元のプログラムを再実行したときなど (第 1.3.3 項を参照) に再度有効にすることもできます。無効にされたウォッチポイントは，SHOW WATCHコマンドを実行したときに表示されるリストでは無効なものとして表示されます。

ウォッチポイントを有効にするには，ACTIVATE WATCHコマンドを使用します。このコマンドを使用すれば，プログラムの実行中にウォッチポイントが有効になります。静的ウォッチポイントはいつでも有効にできますが，その変数が定義されている有効範囲外に実行が移ると，非静的なウォッチポイントは取り消されます (第 3.4.3 項を参照)。

DEACTIVATE WATCH/ALLコマンドおよびACTIVATE WATCH/ALLコマンドは，すべてのウォッチポイントに対し作用し，特に RERUN コマンドを使用してプログラムを再実行するときに有効です。

ウォッチポイントを取り消すには，CANCEL WATCHコマンドを使用します。取り消されたウォッチポイントは，SHOW WATCHコマンドで表示されるリストには載らなくなります。

3.4.2 ウォッチポイント・オプション

SET WATCH コマンドには， SET BREAK コマンドおよび SET TRACE コマンドでのブレークポイントおよびトレースポイント用のオプションと同じオプションがあり，これらのオプションによってウォッチポイントでのデバッガの動作を制御しています。すなわち，/AFTER，/[NO]SILENT，/[NO]SOURCE，および/TEMPORARY の各修飾子とオプションの WHEN 句と DO 句です。これらの使用例は，第 3.3.4 項を参照してください。

3.4.3 非静的変数のウォッチ

注意

ここでは総称して非静的変数という用語を使用しますが，ある言語では自動変数と呼ぶ場合もあります。

プログラム変数の記憶領域は，静的または非静的に割り当てられます。静的変数はプログラムの実行が終了するまで，常に同じメモリ・アドレスに対応づけられます。非静的変数は，呼び出しスタック上またはレジスタ内に割り当てられます。また，非静的編集が値を持つのは，その変数を定義しているルーチンがアクティブなときだけ，つまり呼び出しスタック上にあるときだけです。ここで説明するように，ウォッチポイントの設定方法，ウォッチポイントの動作，およびプログラムの実行速度は，その変数が静的変数か非静的変数かによって異なります。

変数がどのように割り当てられているかを知るには， EVALUATE/ADDRESS コマンドを使用します。静的変数は通常， P0 空間内 (16 進数で 0 から 3FFFFFFF まで) にアドレスを持ちます。非静的変数は通常， P1 空間内 (16 進数で 40000000 から 7FFFFFFF まで) にアドレスを持つか，またはレジスタ内に格納されます。次の Pascal のコード例では， X が静的変数として宣言されています。一方， Y は省略時の設定によって非静的変数です。 EVALUATE/ADDRESS コマンドがデバッグ中に入力され， X はメモリ記憶位置 512 に， Y は R0 レジスタ内にそれぞれ割り当てられていることが示されています。

```
   .
   .
   .
VAR
     X: [STATIC] INTEGER;
     Y: INTEGER;
   .
   .
   .

DBG> EVALUATE/ADDRESS X
512
DBG> EVALUATE/ADDRESS Y
%R0
DBG>
```

SET WATCH コマンドを使用する場合，次の違いに注意してください。静的変数にウォッチポイントを設定するのは，プログラムの実行中ならいつでも可能ですが，非静的変数にウォッチポイントを設定できるのは，その変数を定義しているルーチンの有効範囲内で実行が停止しているときだけです。それ以外の場合には，デバッガは次のような警告を発行します。

```
DBG> SET WATCH Y
%DEBUG-W-SYMNOTACT, nonstatic variable 'MOD4\ROUT3\Y'
    is not active
DBG>
```

非静的変数にウォッチポイントを設定する方法については，第 3.4.3.2 項を参照してください。

3.4.3.1 実行速度

ウォッチポイントを設定した場合，プログラムの実行速度は変数が静的変数であるか非静的変数であるかによって異なります。静的変数のウォッチでは，デバッガはその変数を含むページを書き込み禁止にします。プログラムがそのページに書き込もうとすると (その変数の値を変更しようとすると)，アクセス違反エラーが発生し，デバッガが例外を処理します。すなわち，一時的にそのページに対する書き込み禁止を解除して，命令を終了させ，ウォッチされている変数が変更されたかどうかを決定します。そのページに書き込むとき以外は，プログラムは最大速度で実行されます。

呼び出しスタックまたはレジスタを書き込み禁止にすると問題が発生するので，デバッガは，非静的変数をウォッチするために別の方法を使用します。すなわち，その変数を定義しているルーチン内の各命令をトレースし，命令の実行後，変数値をチェックします。これは，プログラムの実行速度を著しく低下させるので，デバッガは非静的変数にウォッチポイントが設定されると，次のようなメッセージを発行します。

```
DBG> SET WATCH Y
%DEBUG-I-WPTTRACE, nonstatic watchpoint, tracing every instruction
DBG>
```

3.4.3.2 非静的変数へのウォッチポイントの設定

非静的変数にウォッチポイントを設定する場合，その変数を定義しているルーチン内で実行が停止していることを確認します。これを簡単に行うには，そのルーチンにトレースポイントを設定して，さらに変数にウォッチポイントを設定する DO 句を指定します。こうしておけば，そのルーチンが呼ばれるたびにトレースポイントが検出され，ルーチン内のローカル変数に対して自動的にウォッチポイントが設定されます。次の例では， WPTTRACE メッセージによって，ウォッチポイントがルーチン ROUT3 のローカル非静的変数である Y に設定されたことが分かります。

```
DBG> SET TRACE/NOSOURCE ROUT3 DO (SET WATCH Y)
DBG> GO
   .
   .
   .
trace at routine MOD4\ROUT3
%DEBUG-I-WPTTRACE, nonstatic watchpoint, tracing every instruction
   .
   .
   .
watch of MOD4\ROUT3\Y at MOD4\ROUT3\%LINE 16
   16:     Y := 4
   old value:    3
   new value:    4
break at MOD4\ROUT3\%LINE 17
   17:     SWAP(X,Y);
DBG>
```

実行が，ルーチン ROUT3 の呼び出し元に復帰すると，変数 Y はアクティブでなくなります。したがって，デバッガは自動的にウォッチポイントを取り消し，次のようなメッセージを発行します。

```
%DEBUG-I-WATCHVAR, watched variable MOD4\ROUT3\Y has gone out of scope
%DEBUG-I-WATCHCAN, watchpoint now canceled
```

3.4.3.3 非静的変数のウォッチ・オプション

SET WATCH コマンドの修飾子/OVER，/INTO，および/[NO]STATIC には，非静的変数をウォッチするときのオプションがあります。

非静的変数にウォッチポイントを設定する場合，ルーチン呼び出し時に，次の 2 つのうちいずれかを実行するようデバッガに指示できます。

- 呼び出されたルーチンを 1 ステップとして最大速度で実行し，呼び出し 元に復帰後，命令のトレースを再開する。 これは，省略時の設定である (SET WATCH /OVER)。
- 呼び出されたルーチン内で命令をトレースする。すなわち，ルーチン 内で命令が実行されるたびに，変数をモニタする (SET WATCH/INTO)。

SET WATCH/OVER コマンドを使用したほうが，性能が良くなります。ただし，呼び出されたルーチンがウォッチされている変数を変更した場合は，実行が呼び出し元に戻ってから，ウォッチポイントが検出されます。 SET WATCH/INTO コマンドは，プログラムの実行速度を低下させますが，呼び出されたルーチン内でより正確にウォッチポイントをモニタできます。

デバッガは，変数のアドレス (P0 空間， P1 空間，またはレジスタ) を見ることによって，その変数が静的変数であるか非静的変数であるかを決定します。ユーザは， SET WATCH コマンドの入力時に/[NO]STATIC 修飾子を指定することで，この決定を上書きすることができます。たとえば， P1 空間内に非スタック領域を割り当てた場合は， SET WATCH/STATIC コマンドを使用して，その変数は P1 空間内に存在しても

静的変数であることを指定します。反対に，自分自身の呼び出しスタックを P0 空間内に割り当てた場合には， SET WATCH/NOSTATIC コマンドを使用して，その変数は P0 空間内に存在しても非静的変数であることを指定します。

3.4.3.4 インストールされた書き込み可能な共用可能イメージへのウォッチポイントの設定

インストールされた書き込み可能な共用可能イメージにウォッチポイントを設定するには， SET WATCH/NOSTATIC コマンドを使用します (第 3.4.3.3 項を参照)。

非静的なウォッチポイントを設定しなければならない理由は，次のとおりです。このような共用可能イメージ内で宣言された変数は通常，静的変数です。省略時の設定では，デバッガは，その変数を含むページを書き込み禁止にすることによって静的変数をウォッチします。しかし，インストールされた書き込み可能な共用可能イメージ内のページを，書き込み禁止にすることはできません。したがって，デバッガは，非静的変数の場合と同じように，変数値の変更を発見するために性能の良くない手法を使用しなければなりません。すなわち，個々の命令が実行されるたびに，ウォッチされている記憶位置の値を調べる方法です (第 3.4.3.1 項を参照)。

他のプロセスが，ウォッチされている記憶位置の値を変更したとしても，デバッガは，ユーザのプログラムがその値を変更したと報告する可能性があります。

4
プログラム・データの検査と操作

本章では，任意のプログラム記憶位置の内容とプログラム内で宣言したシンボルの値を表示したり変更したりするための EXAMINE コマンドと DEPOSIT コマンドの使用法について説明します。また，言語式を評価する EVALUATE などのコマンドの使用法についても説明します。

本章には次の内容が含まれています。

- EXAMINE， DEPOSIT， EVALUATE の各コマンドの使用に関連した概要。
- シンボリック名 (たとえば，プログラム内で宣言した変数名やルーチン名など) を伴うコマンドの使用。そのようなシンボリック・アドレス式はコンパイラ生成型に対応づけられる。
- シンボリック名を持たず，プログラム記憶位置 (メモリ・アドレスまたはレジスタ) を伴うコマンドの使用。そのようなアドレス式はコンパイラ生成型には対応づけられない。
- アドレス式に対応づけられた型を上書きするための型の指定。

本章の例には言語固有のすべての動作が含まれているわけではありません。どの言語でデバッグを行う場合でも，次の説明を必ず参照してください。

- 第 14.3 節。ここでは，複数言語プログラムをデバッグする場合に注意しなければならない重要な言語の相違点を詳しく説明している。
- デバッガのオンライン・ヘルプ (HELP Languageと入力する)。
- その言語といっしょに提供されたドキュメント。

4.1 概要

この節では EXAMINE， DEPOSIT， EVALUATE の各コマンドを紹介し，これらのコマンドに共通した概要を説明します。

4.1.1 デバッグ時の変数へのアクセス

注意

ここでは総称して非静的変数という用語を使用しますが，ある言語では自動変数と呼ぶ場合もあります。

非静的 (スタック・ローカルまたはレジスタ) 変数を検査したり，そこに値を格納したりするには，その変数を定義しているルーチンが呼び出しスタック上でアクティブでなければなりません。すなわち，定義しているルーチン内のどこかでプログラムの実行が一時停止していなければなりません。非静的変数についての詳しい説明は，第 3.4.3 項を参照してください。

静的変数はプログラム実行中にいつでも検査でき，非静的変数は，それを定義しているルーチンへ到達したときに検査できます。しかし，変数を検査する前に，それが宣言され初期化されている場所を越えてプログラムを実行するようにしてください。初期化されていない変数に含まれる値は無効とみなさなければなりません。

多くのコンパイラでは，プログラムの実行速度を向上させるためにコードを最適化します。デバッグ中のコードが最適化されたものである場合は，プログラム記憶位置がソース・コードから予想したものと一致しないことがあります。特に，最適化技法の中には，特定の変数を排除するものがあるので，デバッグ時にそれらの変数へはアクセスできなくなります。

第 14.1 節には，実行可能コードに対するいくつかの最適化技法の効果が説明されています。最初にプログラムをデバッグする場合は，できれば/NOOPTIMIZE またはそれに相当するコンパイラ・コマンド修飾子を使用して最適化を禁止するのが最善の方法です。

EXAMINE コマンドまたは DEPOSIT コマンドを変数名，またはその他のシンボリック・アドレス式といっしょに使用する場合，モジュールの設定や有効範囲かパス名の指定が必要になることがあります。それらの概念は第 5 章で説明します。本章の例では，すべてのモジュールが設定され，すべての変数名が固有に定義されることと想定しています。

4.1.2 EXAMINE コマンドの使用

高級言語プログラムでは， EXAMINE コマンドはほとんどの場合，変数の現在の値を表示するために使用され，次の構文をとります。

```
EXAMINE address-expression[, . . . ]
```

たとえば次のコマンドは整変数 X の現在の値を表示します。

```
DBG> EXAMINE X
MOD3\X:   17
DBG>
```

値を表示する場合，デバッガは変数名の前にパス名 (この場合は変数 X が宣言されているモジュールの名前) を付けます (第 5.3.2 項を参照)。

一般的には，EXAMINE コマンドはアドレス式で表された要素の現在の値を，その記憶位置に対応づけられた型 (たとえば，整数，実数，配列，レコードなど) で表示します。

EXAMINE コマンドを入力すると，デバッガはプログラム記憶位置 (メモリ・アドレスまたはレジスタ) を得るためにアドレス式を評価します。その後，デバッガはその記憶位置に格納されている値を次のように表示します。

- 記憶位置にシンボリック名がある場合，デバッガはそのシンボルに対応したコンパイラ生成型に従って値を編集する。
- 記憶位置にシンボリック名がない場合，デバッガは省略時の設定では値をロングワード整数型として編集する。

シンボリック・アドレス式および非シンボリック・アドレス式に対応する型についての詳しい説明は，第 4.1.5 項を参照してください。

省略時の設定では，デバッガは値を表示する場合にシンボル情報が入手できるのであれば，アドレス式とそのパス名をシンボルによって示します。アドレスのシンボル化についての詳しい説明は第 4.1.11 項を参照してください。

デバッガは，次のように wchar_t 変数を直接検査することができます。

```
DBG> EXAMINE wide_buffer
TST\main\wide_buffer[0:31]: 'test data line 1.............'
```

Integrity サーバ上の OpenVMS Debugger は，レジスタ・リネーム・ベース (CFM.rrb) とローテート・サイズ (CFM.sor) が両方ゼロであるかのように汎用レジスタ，浮動小数点レジスタ，およびプリディケート・レジスタを表示します。つまり，ローテーションしようとしているレジスタが使用中の場合，そのローテーションは無視されます。

注意

このような状況は，C++ やアセンブリング言語プログラムで稀に発生することがあります。ほとんどのプログラムは影響を受けません。

このような場合は，CFM レジスタを検査し，ゼロでない CFM.rrb および CFM.sor フィールドを検査するように EXAMINE コマンドを手動で調整します。

4.1.3 DUMP コマンドの使用

DCL の DUMP コマンドと同じ方法でメモリの内容を表示するには，次のいずれかの形式でデバッガの DUMP コマンドを使用します。

Binary
Byte

Decimal
Hexadecimal
Longword (省略時の設定)
Octal
Quadword
Word

DUMP コマンドの構文は次のとおりです。

```
DUMP address-expression1[:address-expression2]
```

address-expression2 の省略時の設定は*address-expression1* です。たとえば，次のコマンドはレジスタ R16 ～ R25 の現在の値をクォドワード形式で表示します。

```
DBG> DUMP/QUADWORD R16:R25
 0000000000000078 0000000000030038 8.......x....... %R16
 000000202020786B 0000000000030041 A.......kx   ... %R18
 0000000000030140 0000000000007800 .x......@....... %R20
 0000000000010038 0000000000000007 ........8....... %R22
 0000000000000006 0000000000000000 ................ %R24
DBG>
```

DUMP コマンドを使用すると，レジスタ，変数，配列の内容を表示できます。デバッガは配列の構造体を解釈しません。次の修飾子は，デバッガが DUMP コマンドからの出力を表示する方法を指定します。

修飾子	出力の形式
/BINARY	2 進整数
/BYTE	1 バイトの整数
/DECIMAL	10 進整数
/HEXADECIMAL	16 進整数
/LONGWORD	ロングワード整数 (4 バイト長)
/OCTAL	8 進整数
/QUADWORD	クォドワード整数 (8 バイト長)
/WORD	ワード整数 (2 バイト長)

省略時の設定では，デバッガは，確認した値がコンパイラで生成されるデータ型でない場合，それをロングワードとして表示します。

4.1.4 DEPOSIT コマンドの使用

高級言語では， DEPOSIT コマンドはほとんどの場合，変数に新しい値を代入するために使用されます。このコマンドは，多くのプログラミング言語の代入文と似ていて，次の構文をとります。

```
DEPOSIT address-expression = language-expression
```

たとえば次の DEPOSIT コマンドは整変数 X に値 23 を代入します。

```
DBG> EXAMINE X
MOD3\X:   17
DBG> DEPOSIT X = 23
DBG> EXAMINE X
MOD3\X:   23
DBG>
```

一般的には， DEPOSIT コマンドは言語式を評価し，その結果の値をアドレス式で表されたプログラム記憶位置に格納します。

DEPOSIT コマンドを入力すると，デバッガは次のことを行います。

- プログラム記憶位置を得るためにアドレス式を評価する。
- プログラム記憶位置にシンボリック名がある場合，デバッガはその記憶位置をそのシンボルのコンパイラ生成型に対応づける。記憶位置にシンボリック名がない場合，デバッガは，省略時の設定では，その記憶位置をロングワード整数型に対応づける (第 4.1.5 項を参照)。
- 値を得るため，現在の言語の構文および現在の基数で言語式を評価する。この動作は EVALUATE コマンドの動作と同じである (第 4.1.6 項を参照)。
- 言語式の値と型がアドレス式の型と適合するかどうかを調べる。アドレス式の型と互換性のない値を格納しようとした場合，デバッガは診断メッセージを発行する。値が互換性を持つ場合，デバッガはその値をアドレス式で表された記憶位置に格納する。

その言語の規則で許されていれば，デバッガは格納操作時に型変換を行うことがあるので注意してください。たとえば，X が整変数である場合，次の例では実数値 2.0 は整数値 2 へ変換され，その後で X に代入されます。

```
DBG> DEPOSIT X = 2.0
DBG> EXAMINE X
MOD3\X:   2
DBG>
```

通常，デバッガは現在の言語の代入規則に従おうとします。

4.1.5 アドレス式とそれに対応した型

プログラム内で宣言したシンボル (変数名，ルーチン名など) は，シンボリック・アドレス式です。それらはメモリ・アドレスかレジスタを表します。シンボリック・アドレス式 (本章では「シンボリック名」とも呼ぶ) はコンパイラ生成型を持ち，デバッガはシンボリック名に対応する型と記憶位置が分かっています。シンボリック名からメモリ・アドレスとレジスタ名を取得する方法と，プログラム記憶位置をシンボル化する方法については，第 4.1.11 項を参照してください。

シンボリック名には次のカテゴリがあります。

- 変数

  対応するプログラム記憶位置には変数の現在の値が入っています。変数を検査する方法とそこに値を格納する方法については，第 4.2 節で説明します。

- ルーチン，ラベル，行番号

  対応するプログラム記憶位置には命令が入っています。命令を検査する方法とそこに値を格納する方法については，第 4.3 節で説明します。

シンボリック名を持たないプログラム記憶位置は，コンパイラ生成型に対応づけられません。そのような記憶位置を検査したりそこに値を格納したりできるようにするため，デバッガはそれらの記憶位置を省略時の型であるロングワード整数へ対応づけます。シンボリック名を持たない記憶位置を指定した場合には， EXAMINE コマンドは指定されたアドレスから始まる 4 バイトの内容を表示し，表示される情報を整数値として編集します。次の例では，メモリ・アドレス 926 はシンボリック名に対応づけられていません (EXAMINE コマンドを実行した場合，このアドレスはシンボル化されないことに注意してください)。したがって， EXAMINE コマンドはそのアドレスの値をロングワード整数として表示します。

```
DBG> EXAMINE 926
926:  749404624
DBG>
```

省略時の設定では，シンボリック名を持たないプログラム記憶位置へ最高 4 バイトの整数データを格納できます。このデータはロングワード整数として編集されます。次に例を示します。

```
DBG> DEPOSIT 926 = 84
DBG> EXAMINE 926
926:  84
DBG>
```

シンボリック名を持たない記憶位置を検査する方法とそこへ値を格納する方法については，第 4.5 節で説明します。

EXAMINE コマンドと DEPOSIT コマンドには型修飾子 (/ASCII:*n*,/BYTE など) が使用でき，これらを使用すればプログラム記憶位置と対応した型を上書きすることができます。これは，記憶位置の内容を別の型で解釈し表示する場合や，特定の型の値を別の型に対応づけられた記憶位置に格納したい場合のどちらにも役立ちます。型を上書きする方法については第 4.5 節で説明します。

4.1.6 言語式の評価

言語式は， 1 つまたは複数のシンボル，リテラル，および演算子を組み合わせたものからなり，現在の言語の構文と現在の基数で 1 つの値として評価されます。現在の言語と現在の基数は，それぞれ第 4.1.9 項と第 4.1.10 項に定義されています。次のデバッガ・コマンドと構造は言語式を評価します。

- EVALUATE コマンドと DEPOSIT コマンド。これらはそれぞれ本項と第 4.1.4 項で説明されている。
- IF， FOR， REPEAT， WHILE の各コマンド (第 13.6 節を参照)。
- WHEN 句。これは SET BREAK， SET TRACE， SET WATCH の各コマンドといっしょに使用される (第 3.3.4 項を参照)。

この説明は言語式を評価するすべてのコマンドと構造に対して当てはまりますが，特に EVALUATE コマンドの使用を念頭に置いています。

EVALUATE コマンドは 1 つまたは複数の言語式を現在の言語の構文と現在の基数で評価し，その結果の値を表示します。このコマンドは次の形式をとります。

```
EVALUATE language-expression[, . . . ]
```

EVALUATE コマンドの使用例の 1 つは，プログラムに関連のない算術演算を行うために使用するものです。次に例を示します。

```
DBG> EVALUATE (8+12)*6/4
30
DBG>
```

デバッガは，言語式を評価する場合，現在の言語の演算子プロシージャの規則を使用します。

変数とその他の構造を含む言語式を評価することもできます。たとえば，次の EVALUATE コマンドは整変数 X の現在の値から 3 を差し引き，その結果に 4 を掛けた結果の値を表示します。

```
DBG> DEPOSIT X = 23
DBG> EVALUATE (X - 3) * 4
80
DBG>
```

しかし，関数呼び出しを含む言語式を評価することはできません。たとえば， PRODUCT が 2 つの整数を掛け合わせる関数である場合に， EVALUATE PRODUCT(3,5) コマンドを入力することはできません。プログラムで関数の戻り値を変数に代入する場合は，その変数の結果の値を検査することができます。

式にさまざまなコンパイラ生成型のシンボルが入っている場合，デバッガは式を評価するために現在の言語の型変換規則を使用します。型に互換性がなければ，診断メッセージが発行されます。言語式内での演算子とその他の構造のデバッガ・サポートはデバッガのオンライン・ヘルプに各言語ごとにリストしてあります (HELP Languageを入力します)。

組み込みシンボル%CURVAL は**現在の値**(EVALUATE コマンドまたは EXAMINE コマンドが最後に表示した値，あるいは DEPOSIT コマンドによって格納された値) を表します。バックスラッシュ(\ )もまた，その状況で使用された場合は現在の値を表します。次に例を示します。

```
DBG> EXAMINE X
MOD3\X:  23
DBG> EVALUATE %CURVAL
23
DBG> DEPOSIT Y = 47
DBG> EVALUATE \
47
DBG>
```

4.1.6.1 言語式での変数の使用

言語式内では，プログラムのソース・コード内で変数を使用するのとほぼ同じように変数を使用できます。

したがって，デバッガは通常，言語式内で使用された変数をその変数のアドレスとしてではなく，その変数の現在の値として解釈します。次に例を示します (X は整変数)。

```
DBG> DEPOSIT X = 12     ! 値 12 を X に代入する。
DBG> EXAMINE X          ! X の値を表示する。
MOD4\X:  12
DBG> EVALUATE X         ! X の値を評価して表示する。
12
DBG> EVALUATE X + 4     ! X の値に 4 を加える。
16
DBG> DEPOSIT X = X/2    ! X の値を 2 で割り，その結果の値を
                              ! X に代入する。
DBG> EXAMINE X          ! X の新しい値を表示する。
MOD4\X:   6
DBG>
```

上記の例で示したような言語式内での変数の使用は，通常，単一値の非複合変数に限られます。通常，複数値の複合変数 (配列やレコードなど) を言語式内で指定できるのは，その構文が単一値 (集合体の 1 要素) だけを参照するための構文である場合だけです。たとえば， ARR が整数配列の名前である場合，次のコマンドは無効です。

```
DBG> EVALUATE ARR
%DEBUG-W-NOVALUE, reference does not have a value
DBG>
```

しかし，次のコマンドは配列の 1 要素だけを参照しているので有効です。

```
DBG> EVALUATE ARR(2)          ! 配列 ARR の要素 2 を評価する。
37
DBG> DEPOSIT K = 5 + ARR(2)  ! 2 つの整数値の合計を
DBG>                              ! 1 つの整変数に格納する。
```

現在の言語が BLISS の場合，デバッガは言語式内の変数をその変数のアドレスとして解釈します。変数内に格納されている値を示すには，内容演算子 (ピリオド(.)) を使用しなければなりません。たとえば，言語が BLISS に設定されている場合は次のとおりです。

```
DBG> EXAMINE Y          ! Y の値を表示する。
MOD4\Y:  3
DBG> EVALUATE Y         ! Y のアドレスを表示する。
02475B
DBG> EVALUATE .Y        ! Y の値を表示する。
3
DBG> EVALUATE Y + 4     ! Y のアドレスに 4 を加算し
02475F                      ! その結果の値を表示する。
DBG> EVALUATE .Y + 4    ! Y の値に 4 を加算し
7                           ! その結果の値を表示する。
DBG>
```

どの言語の場合も，変数のアドレスを取得するには，第 4.1.11 項に述べるように EVALUATE/ADDRESS コマンドを使用します。 EVALUATE コマンドと EVALUATE/ADDRESS コマンドは，言語が BLISS に設定してある場合，どちらもアドレス式のアドレスを表示します。

4.1.6.2 デバッガによる数値の型変換

精度が異なる複数の数値型が入っている言語式を評価する場合，デバッガは評価の前にまず精度が低いほうの型を高い精度へ変換します。次の例では，デバッガは加算を行う前に整数 1 を実数の 1.0 へ変換します。

```
DBG> EVALUATE 1.5 + 1
2.5
DBG>
```

基本的な規則は次のとおりです。

- 整数型と実数型が混在する場合，整数型が実数型へ変換される。
- サイズが異なる整数型 (たとえば，バイト整数とワード整数) が混在する場合，サイズが小さいほうの型がサイズの大きいほうの型へ変換される。
- サイズが異なる実数型 (たとえば， S 浮動小数点数型と T 浮動小数点数型) が混在する場合，サイズが小さいほうの型がサイズの大きいほうの型へ変換される。

通常，デバッガではプログラミング言語より多様な数値の型変換ができます。また，デバッガの計算に使用されるハードウェア型 (ワード，ロングワード， S 浮動小数点数など) が，コンパイラの選択する型と異なる場合があります。デバッガは一部の言語ほど型指定が強力でなく厳密でもないので， EVALUATE コマンドによる式の評価が，コンパイラ生成コードによって計算され EXAMINE コマンドによって得られる結果と異なる場合もあります。

4.1.7 言語式と比較した場合のアドレス式

アドレス式を言語式と混同してはなりません。アドレス式がプログラム記憶位置を指定するのに対し，言語式は値を指定します。特に， EXAMINE コマンドはアドレス式をパラメータとして想定し， EVALUATE コマンドは言語式をパラメータとして想定します。これらの点を次の例に示します。

この例では，変数 X に値 12 が格納されます。これは EXAMINE コマンドによって確認されます。 EVALUATE コマンドは，現在の X の値と整数リテラル 6 の和を計算して表示します。

```
DBG> DEPOSIT X = 12
DBG> EXAMINE X
MOD3\X: 12
DBG> EVALUATE X + 6
18
DBG>
```

次の例では， EXAMINE コマンドは， X のアドレスを 6 バイト超えたメモリ記憶位置に現在格納されている値を表示します。

```
DBG> EXAMINE X + 6
MOD3\X+6: 274903
DBG>
```

この場合，記憶位置はコンパイラ生成型に対応づけられていません。したがって，デバッガはその記憶位置に格納されている値をロングワード整数型で解釈し表示します (第 4.1.5 項を参照)。

次の例では， X + 6 (つまり 18) の値が X のアドレスを 6 バイト超えた記憶位置に格納されます。これは最後の EXAMINE コマンドによって確認されます。

```
DBG> EXAMINE X
MOD3\X: 12
DBG> DEPOSIT X + 6 = X + 6
DBG> EXAMINE X
MOD3\X: 12
DBG> EXAMINE X + 6
MOD3\X+6: 18
DBG>
```

4.1.8 現在の値，前の値，および次の値の指定

EXAMINE コマンドと DEPOSIT コマンドを使用する場合，現在と前と次の各データ記憶位置 (論理要素) を高速に参照するため， 3 つの特殊な組み込みシンボル (アドレス式) が使用できます。それらは，ピリオド(.)とサーカンフレックス(^)と Return キーです。

EXAMINE コマンドまたは DEPOSIT コマンドといっしょにピリオド(.)だけを使用した場合，それは現在の値 (EXAMINE コマンドまたは DEPOSIT コマンドによって参照された最新のプログラム記憶位置) を表します。次に例を示します。

```
DBG> EXAMINE X
SIZE\X: 7
DBG> DEPOSIT . = 12
DBG> EXAMINE .
SIZE\X: 12
DBG>
```

サーカンフレックス(^)と Return キーはそれぞれ，前と次の論理データ記憶位置を最新の EXAMINE コマンドまたは DEPOSIT コマンドとの相対関係で表します (それぞれ，論理的先行データと論理的後続データに相当します)。サーカンフレックスと Return キーは，配列の連続した添字付き要素を参照するのに役立ちます。次の例は，これらの演算子の使用法を整数配列 ARR を使用して示しています。

```
DBG> EXAMINE ARR(5)     ! 配列 ARR の要素 5 を検査する。
MAIN\ARR(5): 448670
DBG> EXAMINE ^          ! 前の要素 (4) を検査する。
MAIN\ARR(4): 792802
DBG> EXAMINE Return     ! 次の要素 (5) を検査する。
MAIN\ARR(5): 448670
DBG> EXAMINE Return     ! 次の要素 (6) を検査する。
MAIN\ARR(6): 891236
DBG>
```

デバッガは，論理的後続データと論理的先行データを判別するために現在の値に対応する型を使用します。

ピリオド，サーカンフレックス，および Return キーと同じ目的で，組み込みシンボルの%CURLOC，%PREVLOC，および%NEXTLOC を使用することもできます。これらのシンボルはコマンド・プロシージャ内で役立つほか，プログラムでサーカンフレックスを別の目的に使用する場合に便利です。さらに，論理的後続データを示すために Return キーを使用することは，すべての状況に適用できるわけではありません。たとえば， DEPOSIT コマンドを入力したあとで次の記憶位置を示すために Return キーを押すことはできませんが，シンボル%NEXTLOC ならば同じ目的で常に使用することができます。

EXAMINE コマンドおよび DEPOSIT コマンドと同様に， EVALUATE/ADDRESS コマンドも現在と前と次の論理要素組み込みシンボルの値を再設定します (第 4.1.11 項を参照)。ただし， EVALUATE/ADDRESS コマンドを入力したあとに，次の記憶位置を示すために Return キーを押すことはできません。デバッガの組み込みシンボルについての詳しい説明は，付録 B を参照してください。

上記の例は， EXAMINE コマンドまたは DEPOSIT コマンドでシンボリック名を参照したあとの組み込みシンボルの使用を示したものです。メモリ・アドレスの検査やそこへの値の格納を行う場合，その記憶位置がコンパイラ生成型に対応づけられている場合もそうでない場合もあります。メモリ・アドレスを参照する場合，デバッガは論理的先行データと論理的後続データを判別するために次の規則を使用します。

- アドレスがシンボリック名 (変数，複合変数の構成要素，ルーチンなどの名前) を持っている場合，デバッガはそれに対応したコンパイラ生成型を使用する。
- アドレスがシンボリック名を持っていない場合，デバッガは省略時の設定ではロングワード整数型を使用する。

現在の値が新しい検査操作または格納操作によって再設定されたとき，デバッガは，論理的後続データおよび論理的先行データを判別するために指示された方法で新しい記憶位置を型に対応づけます。次に例を示します。

FORTRAN プログラムで 3 つの変数， ARY， FLT，および BTE を次のように宣言したと想定します。

- ARY は 3 つのワード整数 (各 2 バイト) の配列である。
- FLT は F 浮動小数点数型 (4 バイト) である。
- BTE はバイト整数 (1 バイト) である。

これらの変数用の記憶域が，メモリ内の 1000 から始まる連続したアドレスに割り当てられたと想定します。次に例を示します。

```
1000: ARY(1)
1002: ARY(2)
1004: ARY(3)
1006: FLT
1010: BTE
1011: undefined
   .
   .
   .
```

連続した論理データ記憶位置を検査すると，次の結果が得られます。

```
DBG> EXAMINE 1000             ! 1000 に対応した ARY(1) を検査する。
MOD3\ARY(1):  13            ! 現在の値はこの時点で ARY(1) となる。
DBG> EXAMINE                  ! 次の記憶位置 ARY(2) を検査する。
MOD3\ARY(2):   7            ! その際，ARY(1) の型を参照する。
DBG> EXAMINE                  ! 次の記憶位置 ARY(3) を検査する。
MOD3\ARY(3):  19            ! 現在の値はこの時点で ARY(3) となる。
DBG> EXAMINE                  ! 1006 の値 (FLT) を検査する。
MOD3\FLT:  1.9117807E+07  ! 現在の値はこの時点で FLT となる。
DBG> EXAMINE                  ! 1010 の値 (BTE) を検査する。
MOD3\BTE:   43            ! 現在の値はこの時点で BTE となる。
DBG> EXAMINE                  ! 1011 の値 (未定義) を検査する。
1011: 17694732                      ! データをロングワード整数として解釈する。
DBG>                                ! 記憶位置はシンボル化されていない。
```

これと同じ原則が， EXAMINE コマンドおよび DEPOSIT コマンドに型修飾子を使用する場合でも適用されます (第 4.5.2 項を参照)。修飾子の指定する型によって要素のデータ境界が判別され，したがって論理的後続データと論理的先行データも判別されます。

4.1.9 言語固有性と現在の言語

デバッガでは，デバッグ・コンテキストをいずれかのサポートされた言語へ設定することができます。現在の言語を設定することにより，デバッガ・コマンド内にユーザが指定する名前，数字，演算子，および式をデバッガが解析し解釈する方法と，データを表示する方法が決まります。

省略時の設定では，メイン・プログラムを含むモジュールの言語が現在の言語となり，それはユーザがプログラムをデバッガの制御下に置いた時点で識別されます。次に例を示します。

```
$ PASCAL/NOOPTIMIZE/DEBUG TEST1
$ LINK/DEBUG TEST1
$ DEBUG/KEEP

          Debugger Banner and Version Number

DBG> RUN TEST1
Language: PASCAL, Module: TEST1
DBG>
```

別の言語で作成されたコードを持つモジュールをデバッグする場合には，新しい言語固有コンテキストを設定するために SET LANGUAGE コマンドを使用できます。第 14.3 節に重要な言語の相違点が説明してあります。言語式内の演算子とその他の構造に関するデバッガ・サポートは，各言語別にデバッガのオンライン・ヘルプに示されています (HELP Languageと入力します)。

4.1.10 整数データを入力または表示するための基数の指定

デバッガは，整数データを 4 つの基数 (10 進数， 16 進数， 8 進数， 2 進数) のいずれかで解釈し表示することができます。ほとんどの言語の場合，省略時の基数は 10 進数です。

Alpha プロセッサでは BLISS と MACRO–32 と MACRO–64 は例外で，これらの省略時の基数は 10 進数です。

次の種類の整数データでは，基数を制御することができます。

- アドレス式内または言語式内で指定するデータ
- EVALUATE コマンドおよび EXAMINE コマンドで表示されるデータ

このほかの種類の整数データでは，基数を制御できません。たとえば，アドレスは SHOW CALLS の表示では常に 16 進数を基数として表示されます。また，各種のコマンド修飾子 (/AFTER:*n*，/UP:*n*など) といっしょに整数*n*を指定する場合には， 10 進数を基数として使用しなければなりません。

基数の制御に使用する方法は，目的によって異なります。以後のすべてのコマンドに対して新しい基数を設定するには， SET RADIX コマンドを使用します。次に例を示します。

```
DBG> SET RADIX HEXADECIMAL
```

このコマンドが実行されたあと，ユーザがアドレス式または言語式の中へ入力するすべてのデータは 16 進数として解釈されます。また， EVALUATE コマンドおよび EXAMINE コマンドによって表示されるすべての整数データも 16 進数を基数として表示されます。

SHOW RADIX コマンドは現在の基数 (省略時の基数，または SET RADIX コマンドによって設定された最新の基数) を表示します。次に例を示します。

```
DBG> SHOW RADIX
input radix: hexadecimal
output radix: hexadecimal
DBG>
```

SHOW RADIX コマンドは**入力基数** (データ入力用) と**出力基数** (データ表示用) の両方を表示します。 SET RADIX コマンドの修飾子/INPUT および/OUTPUT を使用すれば，データの入力用と表示用に別の基数を指定できます。詳しい説明はオンライン・ヘルプの SET RADIX コマンドの説明を参照してください。

省略時の基数を復元するには CANCEL RADIX コマンドを使用します。

次の例では，現在の基数を変更せずに別の基数で整数データを表示させたり入力したりするための方法をいくつか示します。

現在の基数を変更せずに整数データを別の基数へ変換するには， EVALUATE コマンドに基数修飾子 (/BINARY，/DECIMAL，/HEXADECIMAL，/OCTAL) を指定します。次に例を示します。

```
DBG> SHOW RADIX
input radix: decimal
output radix: decimal
DBG> EVALUATE 18 + 5
23                          ! 23 is decimal integer.
DBG> EVALUATE/HEX 18 + 5
00000017                    ! 17 is hexadecimal integer.
DBG>
```

基数修飾子はデータ入力用の基数には影響を及ぼしません。

整変数の現在の値 (または，整数型を持つプログラム記憶位置の内容) を別の基数で表示するには， EXAMINE コマンドに基数修飾子を指定します。次に例を示します。

```
DBG> EXAMINE X
MOD4\X: 4398                ! 4398 は 10 進整数である。
DBG> EXAMINE/OCTAL .        ! X は現在の値である。
MOD4\X: 00000010456         ! 10456 は 8 進整数である。
DBG>
```

1 つまたは複数の整数リテラルを，現在の基数を変更せずに別の基数で入力するには，基数組み込みシンボル%BIN，%DEC，%HEX，%OCT のいずれかを使用します。基数組み込みシンボルは，次にある整数リテラルまたは括弧で囲まれた式内のすべての数値リテラルをそれぞれ 2 進数， 10 進数， 16 進数，または 8 進数として扱うようデバッガに指示します。これらのシンボルはデータ表示用の基数には影響を及ぼしません。次に例を示します。

```
DBG> SHOW RADIX
input radix: decimal
output radix: decimal
DBG> EVAL %BIN 10             ! 2 進整数 10 を評価する。
2                                   ! 2 は 10 進整数である。
DBG> EVAL %HEX (10 + 10)      ! 16 進整数 20 を評価する。
32                                  ! 32 は 10 進整数である。
DBG> EVAL %HEX 20 + 33        ! 20 を 16 進数，33 を 10 進数として扱う。
65                                  ! 65 は 10 進整数である。
DBG> EVAL/HEX %OCT 4672       ! 4672 を 8 進数として扱い 16 進数で表示する。

000009BA                            ! 9BA は 16 進数である。
DBG> EXAMINE X + %DEC 12      ! X のアドレスを 12 (10 進) バイト超えた
MOD3\X+12:  493847            ! 記憶位置を検査する。
DBG> DEPOS J = %OCT 7777777   ! 8 進数値を格納する。
DBG> EXAMINE .                ! その値を 10 進数を基数として表示する。
MOD3\J:  2097151
DBG> EXAMINE/OCTAL .          ! その値を 8 進数を基数として表示する。
MOD3\J:  00007777777
DBG> EXAMINE %HEX 0A34D       ! 記憶位置 A34D(16 進数) を検査する。
SHARE$LIBRTL+4941:  344938193       ! 344938193 は 10 進整数である。
DBG>
```

注意

数字ではなく英字で始まる 16 進整数 (上記の例では A34D) を指定する場合は，その前に 0 を付けます。 0 を付けないと，デバッガはその整数をプログラム内で宣言されたシンボルとして解釈しようとします。

基数組み込みシンボルについての例は，付録 B にもあります。

4.1.11 メモリ・アドレスの取得とシンボル化

変数名，行番号，ルーチン名，またはラベルなどのシンボリック・アドレス式に対応づけられたメモリ・アドレスまたはレジスタ名を判別するには， EVALUATE /ADDRESS コマンドを使用します。次に例を示します。

```
DBG> EVALUATE/ADDRESS X       ! 変数名
2476
DBG> EVALUATE/ADDRESS SWAP    ! ルーチン名
1536
DBG> EVALUATE/ADDRESS %LINE 26
1629
DBG>
```

アドレスは，(第 4.1.10 項で定義したとおり) 現在の基数で表示されます。アドレスを別の基数で表示したい場合は基数修飾子を指定できます。次に例を示します。

```
DBG> EVALUATE/ADDRESS/HEX X
000009AC
DBG>
```

変数がメモリ・アドレスではなくレジスタに対応づけられている場合， EVALUATE /ADDRESS コマンドは，基数修飾子が使用されているかどうかに関係なくレジスタの名前を表示します。次のコマンドは変数 K (非静的変数) がレジスタ R2 に対応することを示します。

```
DBG> EVALUATE/ADDRESS K
%R2
DBG>
```

EXAMINE コマンドおよび DEPOSIT コマンドと同様に， EVALUATE/ADDRESS は現在と前と次の論理要素組み込みシンボルの値を再設定します (第 4.1.8 項を参照)。EVALUATE コマンドとは異なり， EVALUATE/ADDRESS は現在の値の組み込みシンボル%CURVAL およびバックスラッシュ(\ )に影響を及ぼしません。

SYMBOLIZE コマンドの動作は EVALUATE/ADDRESS の動作とは逆ですが，現在，前，または次の論理要素組み込みシンボルに影響を及ぼしません。このコマンドは，メモリ・アドレスまたはレジスタ名をシンボリック表現 (パス名を含む) に変換します。ただし，そのような表現が可能な場合です (第 5 章にシンボル化を制御する方法が説明されています)。たとえば，次のコマンドは変数 K がレジスタ R2 に対応することを示します。

```
DBG> SYMBOLIZE %R2
address MOD3\%R2:
    MOD3\K
DBG>
```

省略時の設定では，シンボリック・モードが有効 (SET MODE SYMBOLIC) になります。したがって，デバッガはシンボルがアドレスに使用できる場合，アドレスをすべてシンボルで表示します。たとえば， EXAMINE コマンドで数値アドレスを指定した場合，シンボリック情報が入手できるのであれば，そのアドレスは次のようにシンボリック形式で表示されます。

```
DBG> EVALUATE/ADDRESS X
2476
DBG> EXAMINE 2476
MOD3\X:  16
DBG>
```

ただし，変数に対応するレジスタを指定した場合， EXAMINE コマンドはそのレジスタ名を変数名に変換しません。次に例を示します。

```
DBG> EVALUATE/ADDRESS K
%R2
DBG> EXAMINE %R2
MOD3\%R2:  78
DBG>
```

SET MODE NOSYMBOLIC コマンドを入力した場合にはシンボリック・モードが禁止され，デバッガはシンボリック名でなく数値アドレスを表示します。シンボル化を禁止した場合，デバッガは数字を名前に変換する必要がないので，コマンドの処理がいくらか速くなることがあります。 EXAMINE コマンドには，単一の EXAMINE コマンドのシンボル化を制御できる/[NO]SYMBOLIC 修飾子があります。次に例を示します。

```
DBG> EVALUATE/ADDRESS Y
512
DBG> EXAMINE 512
MOD3\Y:  28
DBG> EXAMINE/NOSYMBOLIC 512
512:  28
DBG>
```

シンボリック・モードはまた，命令の表示にも影響を及ぼします。

たとえば Integrity サーバでは次のとおりです。

```
DBG> EXAMINE/INSTRUCTION .%PC
HELLO\main\%LINE 8: add        r34=200028, r1
DBG> EXAMINE/NOSYMBOL/INSTRUCTION .%PC
65969:              add         r34 = 200028, r1
DBG>
```

4.2 変数の検査と値の格納

この節の例は EXAMINE コマンドと DEPOSIT コマンドでの変数の使用法を示しています。

言語が使用する変数の型，それらの型の名前，および式の中に各種の型を混在させることができる程度は，言語によって異なります。ここでは次の汎用型について説明します。

- スカラ型 (整数型，実数型，文字型，論理型など)
- 文字列型
- 配列型
- レコード型
- ポインタ (アクセス) 型

高級言語プログラム内の変数を検査および操作する場合の最も重要な関連事項は，デバッガがプログラム内の変数の名前，構文，型制約，有効範囲規則を認識するということです。したがって， EXAMINE コマンドまたは DEPOSIT コマンドで変数を指定する場合，ソース・コードに使用する構文と同じ構文を使用します。デバッガはその構文に従ってデータを処理し，表示します。同様に，変数に値を代入する場合も，デバッガはその言語の型指定規則に従います。ユーザが互換性のない値を格納しようとした場合には，診断メッセージが発行されます。以降の例には，そのような無効な操作とその結果生じる診断も含まれています。

DEPOSIT コマンド (またはその他のコマンド) を使用する場合，次の動作に注意してください。デバッガが重大度 I (情報) の診断メッセージが発行した場合でも，コマンドは実行されます (DEPOSIT コマンドの場合は格納される)。デバッガが違法なコマンド行を強制終了するのは，メッセージの重大度が W (警告) 以上の場合だけです。

言語固有情報についての詳しい説明は，デバッガのオンライン・ヘルプを参照してください (HELP Languageと入力します)。

4.2.1 スカラ型

次の例は， EXAMINE， DEPOSIT， EVALUATE の各コマンドで整数型，実数型，型を使用した例です。

3 つの整変数のリストを検査します。

```
DBG> EXAMINE WIDTH, LENGTH, AREA
SIZE\WIDTH:   4
SIZE\LENGTH:  7
SIZE\AREA:   28
DBG>
```

整数式を格納します。

```
DBG> DEPOSIT WIDTH = CURRENT_WIDTH + 10
DBG>
```

デバッガは，代入する値が変数のデータ型の制約と大きさの制約に適合するかどうかを調べます。次の例は境界外の値 (X は正の整数として宣言されている) を格納しようとした場合です。

```
DBG> DEPOSIT X = -14
%DEBUG-I-IVALOUTBNDS, value assigned is out of bounds at or near DEPOSIT
DBG>
```

1 つの言語式の中に複数の数値型 (精度が異なる整数と実数) を混在させようとした場合，デバッガは通常，その言語の規則に従います。データ型が強力な言語では，そのような混在は好ましくありません。一部の言語では，実数値を整変数に格納することができます。ただし，実数値が整数に変換されます。次に例を示します。

```
DBG> DEPOSIT I = 12345
DBG> EXAMINE I
MOD3\I:  12345
DBG> DEPOSIT I = 123.45
DBG> EXAMINE I
MOD3\I:  123
DBG>
```

1 つの式に複数の数値型が混在する場合，デバッガは第 4.1.6.2 項に述べたような型変換を行います。次に例を示します。

```
DBG> DEPOSIT Y = 2.356     ! Y は G 浮動小数点数型である。
DBG> EXAMINE Y
MOD3\Y: 2.35600000000000
DBG> EVALUATE Y + 3
5.35600000000000
DBG> DEPOSIT R = 5.35E3    ! R は F 浮動小数点数型である。
DBG> EXAMINE R
MOD3\R:  5350.000
DBG> EVALUATE R*50
   267500.0
DBG> DEPOSIT I = 22222
DBG> EVALUATE R/I
   0.2407524
DBG>
```

次の例は，論理型変数を使用した操作を示しています。値 TRUE と FALSE が変数 WILLING と ABLE にそれぞれ代入されます。その後， EVALUATE コマンドでそれらの値の論理積を求めています。

```
DBG> DEPOSIT WILLING = TRUE
DBG> DEPOSIT ABLE = FALSE
DBG> EVALUATE WILLING AND ABLE
False
DBG>
```

4.2.2 ASCII 文字列型

ASCII 文字列の値を表示する場合，デバッガは値をその言語の構文に従って二重引用符(")か一重引用符(')で囲みます。次に例を示します。

```
DBG> EXAMINE EMPLOYEE_NAME
PAYROLL\EMPLOYEE_NAME:    "Peter C. Lombardi"
DBG>
```

文字列値(1文字だけの場合も含む)を文字列変数に格納するには，その値を二重引用符(")か一重引用符(')で囲みます。次に例を示します。

```
DBG> DEPOSIT PART_NUMBER = "WG-7619.3-84"
DBG>
```

文字列がアドレス式の表す記憶位置に収まらない数の ASCII 文字(各1バイト)を持つ場合，デバッガは余分な文字を右から切り捨て，次のメッセージを発行します。

```
%DEBUG-I-ISTRTRU, string truncated at or near DEPOSIT
```

文字列の文字数が少ない場合，デバッガは ASCII スペース文字を挿入することによってその文字列の右側の残りの文字を埋めます。

4.2.3 配列型

配列集合体全体を検査したり，1つの添字付き要素を検査したり，1つの断面(要素の範囲)を検査したりできます。しかし，一度に値を格納できるのは1つの要素だけです。次の例は，配列を使用した代表的な操作を示しています。

次のコマンドは，1次元の整数配列である配列変数 ARRX の全要素の値を表示します。

```
DBG> EXAMINE ARRX
MOD3\ARRX
    (1):     42
    (2):     17
    (3):    278
    (4):     56
    (5):    113
    (6):    149
DBG>
```

次のコマンドは，配列 ARRX の要素4の値を表示します。言語に応じて，添字付き要素を表すために括弧または大括弧が使用されます。

```
DBG> EXAMINE ARRX(4)
MOD3\ARRX(4):   56
DBG>
```

次のコマンドは， ARRX の 1 つの断面における全要素の値を表示します。この断面は要素 2 ～要素 5 の要素範囲で構成されます。

```
DBG> EXAMINE ARRX(2:5)
MOD3\ARRX
    (2):     17
    (3):    278
    (4):     56
    (5):    113
DBG>
```

通常，検査する値の範囲は 2 つの値をコロンで区切って示します (value1:value2)。言語によっては，コロンの代わりに 2 つのピリオド(..)を使用できます。

一度に値を格納できる配列要素は 1 つだけです。 1 つの DEPOSIT コマンドで配列断面または配列集合体全体へ値を格納することはできません。たとえば，次のコマンドは値 53 を ARRX の要素 2 へ格納します。

```
DBG> DEPOSIT ARRX(2) = 53
DBG>
```

次のコマンドは，実数の 2 次元配列 (1 次元あたり 3 つ) である配列 REAL_ARRAY の全要素の値を表示します。

```
DBG> EXAMINE REAL_ARRAY
PROG2\REAL_ARRAY
   (1,1):       27.01000
   (1,2):       31.00000
   (1,3):       12.48000
   (2,1):       15.08000
   (2,2):       22.30000
   (2,3):       18.73000
DBG>
```

境界外の添字値へ格納しようとした場合，デバッガは診断メッセージを発行します。次に例を示します。

```
DBG> DEPOSIT REAL_ARRAY(1,4) = 26.13
%DEBUG-I-SUBOUTBND, subscript 2 is out of bounds, value is 4,
bounds are 1..3
DBG>
```

上記の例では，診断メッセージのレベルが I ですので，格納操作が実行されました。このことは， (1,3) に隣接した配列要素， (2,1) の値が境界外の格納操作によって影響を受けた可能性があることを意味します。

ある配列の複数の構成要素に同じ値を格納するには， FOR または REPEAT などのループ・コマンドを使用できます。たとえば，値 RED を配列 COLOR_ARRAY の要素 1 ～要素 4 へ代入するには次のようにします。

```
DBG> FOR I = 1 TO 4 DO (DEPOSIT COLOR_ARRAY(I) = RED)
DBG>
```

配列要素を 1 ステップずつ処理するには，第 4.1.8 項で説明したとおり，組み込みシンボルの(.)と(^)も使用できます。

4.2.4 レコード型

注意

ここでは，異質なデータ型を要素として持つデータ構造を総称してレコードという用語を使用しますが，これは C 言語では*struct*型と呼ばれます。

レコード集合体全体を検査したり，1つのレコード構成要素を検査したり，複数の構成要素を検査したりできます。しかし，一度に値を格納できる構成要素は 1 つだけです。次の例は，レコードの代表的な操作を示しています。

次のコマンドはレコード変数 PART の全構成要素の値を表示します。

```
DBG> EXAMINE PART
INVENTORY\PART:
    ITEM:     "WF-1247"
    PRICE:     49.95
    IN_STOCK:  24
DBG>
```

次のコマンドは一般的な構文で，レコード PART の構成要素 IN_STOCK の値を表示します。

```
DBG> EXAMINE PART.IN_STOCK
INVENTORY\PART.IN_STOCK: 24
DBG>
```

次のコマンドは，上と同じレコード構成要素の値を COBOL 構文を使用して表示します。言語を COBOL に設定しなければなりません。

```
DBG> EXAMINE IN_STOCK OF PART
INVENTORY\IN_STOCK of PART:
    IN_STOCK:  24
DBG>
```

次のコマンドはレコード PART の 2 つの構成要素の値を表示します。

```
DBG> EXAMINE PART.ITEM, PART.IN_STOCK
INVENTORY\PART.ITEM:     "WF-1247"
INVENTORY\PART.IN_STOCK:  24
DBG>
```

次のコマンドはレコード構成要素 IN_STOCK に値を格納します。

```
DBG> DEPOSIT PART.IN_STOCK = 17
DBG>
```

4.2.5 ポインタ (アクセス) 型

ポインタ変数によって指定される (指し示される) 要素を検査し，その要素に値を格納することができます。また，ポインタ変数を検査することもできます。

たとえば，次の Pascal コードは，実数型の値を指し示すポインタ変数*A*を宣言します。

```
   .
   .
   .
TYPE
     T = ^REAL;
VAR
     A : T;
   .
   .
   .
```

次のコマンドは，ポインタ変数*A*によって指し示される要素の値を表示します。

```
DBG> EXAMINE A^
MOD3\A^:  1.7
DBG>
```

次の例では，*A*によって指し示される要素へ値 3.9 が格納されます。

```
DBG> DEPOSIT A^ = 3.9
DBG> EXAMINE A^
MOD3\A^:  3.9
DBG>
```

ポインタ変数の名前を EXAMINE コマンドに指定する場合，デバッガはその変数が指し示すオブジェクトのメモリ・アドレスを表示します。次に例を示します。

```
DBG> EXAMINE/HEXADECIMAL A
SAMPLE\A: 0000B2A4
DBG>
```

4.3 命令の検査と値の格納

デバッガは命令に対応したアドレス式を認識します。これにより，変数の場合と同じ基本的な手法を使用して命令を検査し，そこに値を格納することができます。

命令レベルでデバッグを行う場合，最初に次のコマンドを入力すると便利な場合があります。このコマンドは省略時のステップ・モードを命令ごとのステップ実行に設定します。

```
DBG> SET STEP INSTRUCTION
DBG>
```

これ以外にも，特定の種類の命令に対してプログラムを実行できるステップ・モードがあります。これらの命令で実行に割り込みをかけるためにブレークポイントを設定することもできます。

さらに，ユーザ・プログラムのデコード済み命令ストリームを表示するために，画面モード機械語命令ディスプレイ (第 7.4.4 項を参照) を使用できます。

4.3.1 命令の検査

命令に対応したアドレス式 (行番号など) を EXAMINE コマンドの中に指定した場合，デバッガはその記憶位置にある最初の命令を表示します。その後，第 4.1.8 項で説明したとおり，現在と次と前の命令 (論理要素) をそれぞれ表示するためにピリオド(.)と Return キーとサーカンフレックス(^)を使用できます。

たとえば， Alpha プロセッサでは次のとおりです

```
DBG> EXAMINE %LINE 12
MOD3\%LINE 12:     BIS     R31,R31,R2
DBG> EXAMINE
MOD3\%LINE 12+4:   BIS     R31,R2,R0  ! 次の命令
DBG> EXAMINE
MOD3\%LINE 12+8:   ADDL    R31,R0,R0  ! 次の命令
DBG> EXAMINE ^
MOD3\%LINE 12+4:   BIS     R31,R2,R0  ! 前の命令
DBG>
```

行番号，ルーチン名，およびラベルは，命令に対応づけられるシンボリック・アドレス式です。また，命令はプログラムの実行時に他の各種のメモリ・アドレスや一定のレジスタに格納される場合があります。

プログラム・カウンタ (PC) は，プログラムが次に実行する命令のアドレスが入っているレジスタです。 EXAMINE .%PC コマンドはその命令を表示します。ピリオド(.)は，アドレス式の直前に使用した場合には"内容"演算子 (アドレス式が指し示す記憶位置の内容) を表します。次の違いに注意してください。

- EXAMINE %PC は現在の PC の値，つまり次に実行される命令のアドレスを表示する。
- EXAMINE .%PC はそのアドレスの内容，つまりプログラムが次に実行する命令を表示する。

前の例に示したとおり，デバッガはアドレス式が命令に対応しているかどうかを認識しています。対応している場合， EXAMINE コマンドはその命令を表示します。/INSTRUCTION 修飾子を使用する必要はありません。/INSTRUCTION 修飾子は，任意のプログラム記憶位置の内容を命令として表示するために使用します。つまり， EXAMINE/INSTRUCTION コマンドを使用すると，デバッガはあらゆるプログラム記憶位置の内容を命令として解釈し編集します (第 4.5.2 項を参照)。

MACRO-32 プログラム内の連続した命令を検査する場合，データの記憶域が命令ストリームの中間に割り当てられていると，デバッガがデータを命令と誤って解釈することがあります。次の例はこの問題を説明したものです。この MACRO-32 コードでは，行 7 の BRB 命令の直後に 2 つのロングワード・データ記憶域が割り当てられています。行番号は説明のために加えたものです。

```
module TEST
     1:         .TITLE  TEST
     2:
     3: TEST$START::
     4:         .WORD   0
     5:
     6:         MOVL    #2,R2
     7:         BRB     LABEL_2
     8:
     9:         .LONG   ^X12345
    10:         .LONG   ^X14465
    11:
    12: LABEL_2:
    13:         MOVL    #5,R5
    14:
    15:         .END    TEST$START
```

次の EXAMINE コマンドは行 6 の開始時の命令を表示します。

```
DBG> EXAMINE %LINE 6
TEST\TEST$START\%LINE 6:  MOVL    S^#02,R2
DBG>
```

次の EXAMINE コマンドは，行 7 で論理的後続データ要素を正しく解釈し，表示します。

```
DBG> EXAMINE
TEST\TEST$START\%LINE 7:  BRB     TEST\TEST$START\LABEL_2
DBG>
```

しかし，次の 3 つの EXAMINE コマンドは， 3 つの論理的後続データを誤って命令として解釈します。

```
DBG> EXAMINE
TEST\TEST$START\%LINE 7+2:  MULF3   S^#11.00000,S^#0.5625000,S^#0.5000000
DBG> EXAMINE
%DEBUG-W-ADDRESSMODE, instruction uses illegal or undefined addressing modes
TEST\TEST$START\%LINE 7+6:  MULD3   S^#0.5625000[R4],S^#0.5000000,@W^5505(R0)
DBG> EXAMINE
TEST$START+12:   HALT
DBG>
```

4.4 レジスタの検査と値の格納

EXAMINE コマンドは，プログラム内でアクセス可能なレジスタの内容を表示します。 DEPOSIT コマンドを使用すると，これらのレジスタの内容を変更できます。レジスタの数と種類は，次の項に示すように，各 OpenVMS プラットフォームにより異なります。

4.4.1 Alpha レジスタの検査と値の格納

Alpha プロセッサでは， Alpha アーキテクチャが 32 個の汎用 (整数) レジスタと 32 個の浮動小数点レジスタを備えており，それらの一部は一時的なアドレスやデータの記憶域として使用されます。表 4–1 に， Alpha レジスタを参照するデバッガ組み込みシンボルを示します。

表 4–1 Alpha レジスタ用のデバッガ・シンボル

シンボル	説明
	Alpha 整数レジスタ
%R0 . . . %R28	レジスタ R0 . . . R28
%FP (%R29)	スタック・フレーム・ベース・レジスタ (FP)
%SP (%R30)	スタック・ポインタ (SP)
%R31	ReadAsZero/Sink (RZ)
%PC	プログラム・カウンタ (PC)
%PS	プロセッサ・ステータス・レジスタ (PS)。 Alpha プロセッサでは，組み込みシンボル%PSL および%PSW は無効です。
	Alpha 浮動小数点レジスタ
%F0 . . . %F30	レジスタ F0 . . . F30
%F31	ReadAsZero/Sink

Alpha プロセッサには次のことが該当します。

- プログラムで同じ名前のシンボルを定義していない場合は，パーセント記号(%)の接頭辞を省略できます。
- R30 レジスタには，値を格納できません。

- R31 レジスタと F31 レジスタには，値を格納できません。これらのレジスタには，恒久的に値 0 が割り当てられています。
- ベクタ・レジスタはありません。

次の例は，レジスタの内容を確認し，値を格納する方法を示しています。

```
DBG> SHOW TYPE          ! Show type for locations without
type: long integer      ! a compiler-generated type.
DBG> SHOW RADIX         ! Identify current radix.
input radix: decimal
output radix: decimal
DBG> EXAMINE %R11       ! Display value in R11.
MOD3\%R11:  1024
DBG> DEPOSIT %R11 = 444 ! Deposit new value into R11.
DBG> EXAMINE %R11       ! Check new value.
R11:  444
DBG> EXAMINE %PC        ! Display value in program counter.
MOD\%PC: 1553
DBG> EXAMINE %SP        ! Display value in stack pointer.
0\%SP:  2147278720
DBG>
```

PC についての詳細は，第 4.3.1 項を参照してください。

プロセッサ・ステータス (Alpha **のみ**)

Alpha プロセッサでは，プロセッサ・ステータス (PS) レジスタの値がプロセッサ・ステータス変数の数を表します。 PS の最初の 3 ビットは，このソフトウェアが使用するために予約されています。これらのビットの値はユーザ・プログラムで制御できます。残りのビット (ビット 4 ～ 64) には特権情報が入っており，ユーザ・モード・プログラムでは変更できません。

次の例は PS の内容を検査する方法を示しています。

```
DBG> EXAMINE %PS
MOD1\%PS:
      SP_ALIGN IPL VMM   CM   IP SW
         48     0   0   USER   0  3
DBG>
```

各ビットの値も含めた PS についての完全な説明は，『Alpha Architecture Reference Manual』を参照してください。

PS 内の情報は別の形式でも表示できます。次に例を示します。

```
DBG> EXAMINE/LONG/HEX %PS
MOD1\%PS:        0000001B
DBG> EXAMINE/LONG/BIN %PS
MOD1\%PS:        00000000 00000000 00000000 00011011
DBG>
```

EXAMINE/PS コマンドは，記憶位置の値を PS 形式で表示します。これは，現在の PS 値と保存された PS 値の組み合わせを検査するのに役立ちます。

4.4.2 Integrity レジスタの検査と値の格納

Integrity プロセッサは， Integrity アーキテクチャの以下のレジスタを備えています。

- 最大 128 個の， 64 ビット汎用レジスタ。
- 最大 128 個の， 82 ビット浮動小数点レジスタ (このデバッガでは，これらのレジスタを，完全なオクタワードとして扱うことができます)。
- 最大 64 個の 1 ビット・プレディケート・レジスタ，最大 8 個の 64 ビット分岐レジスタ，最大 128 個 (アクセスや使用が可能なのは 20 個のみ) のアプリケーション・レジスタ。
- 特殊レジスタ (例: %PC) および仮想レジスタ (例: %RETURN_PC)。

これらのレジスタの大半は，ユーザ・モードでのデバッグで，読み取りおよび書き込みが可能です。ただし，一部のレジスタは書き込み不可です。また，アクセスするために， System Code Debugger (SCD) の構成 (『 OpenVMS System Analysis Tools Manual』を参照) に関連する高い特権が必要なレジスタもあります。

表 4–2 Integrity レジスタのデバッガ・シンボル

シンボル	説明
	Integrity アプリケーション・レジスタ
%KR0 . . . %KR7	カーネル・レジスタ 0 . . . 7
%RSC (%AR16)	レジスタ・スタック・コンフィギュレーション
%BSP (%AR17)	バッキング・ストア・ポインタ
%BSPSTORE (%AR18)	メモリ・ストア用バッキング・ストア・ポインタ
%RNAT (%AR19)	RSE NaT コレクション
%CCV ($AR32)	比較交換での比較値
%UNAT (%AR36)	ユーザ NaT コレクション
%FPSR (%AR40)	浮動小数点ステータス
%PFS (%AR64)	以前のファンクション状態
%LC (%AR65)	ループ・カウント
%EC (%AR66)	エピローグ・カウント
%CSD	コード・セグメント
%SSD	スタック・セグメント

(次ページに続く)

表 4–2 (続き) Integrity レジスタのデバッガ・シンボル

シンボル	説明
	コントロール・レジスタ
%DCR (%CR0)	デフォルト・コントロール
%ITM (%CR1)	インターバル・タイマ・マッチ (SCD でのみ参照可能)
%IVA (%CR2)	割り込みベクタ・アドレス (SCD でのみ参照可能)
%PTA (%CR8)	ページ・テーブル・アドレス (SCD でのみ参照可能)
%PSR (%CR9, %ISPR)	割り込みプロセッサ・ステータス
%ISR (%CR17)	割り込みステータス
%IIP (%CR19)	割り込み命令ポインタ
%IFA (%CR20)	割り込みフォルト・アドレス
%ITIR (%CR21)	割り込み TLB 挿入
%IIPA (%CR22)	割り込み命令前アドレス
%IFS (%CR23)	割り込みファンクション状態
%IIM (%CR24)	割り込み即値
%IHA (%CR25)	割り込みハッシュ・アドレス
%LID (%CR64)	ローカル割り込み ID (SCD でのみ参照可能)
%TPR (%CR66)	タスク・プライオリティ (SCD でのみ参照可能)
%IRR0 . . . %IRR3 (%CR68 . . . %CR71)	外部割り込み要求 0 . . . 3 (SCD でのみ参照可能)
%ITV (%CR72)	インターバル・タイマ (SCD でのみ参照可能)
%PMV (%CR73)	パフォーマンス監視 (SCD でのみ参照可能)
%CMCV (%CR74)	訂正済みマシン・チェック・ベクタ (SCD でのみ参照可能)
%IRR0 および%IRR1 (%CR80 および%CR81)	ローカル・リダイレクション 0:1 (SCD でのみ参照可能)

(次ページに続く)

表 4–2 (続き) Integrity レジスタのデバッガ・シンボル

シンボル	説明
	特殊レジスタ
%IH (%SR0)	インボケーション・ハンドル
%PREV_BSP	以前のバッキング・ストア・ポインタ
%PC (%IP)	プログラム・カウンタ (命令ポインタ | スロット番号)
%RETURN_PC	リターン・プログラム・カウンタ
%CFM	現在のフレーム・マーカ
%NEXT_PFS	前々回のファンクション状態
%PSP	以前のスタック・ポインタ
%CHFCTX_ADDR	コンディション・ハンドリング・ファシリティ・コンテキスト・アドレス
%OSSD	オペレーティング・システム固有データ
%HANDLER_FV	ハンドラ・ファンクション値
%LSDA	言語固有データ領域
%UM	ユーザ・マスク
	プレディケート・レジスタ
%PR (%PRED)	プレディケート・コレクション・レジスタ —%P0 . . . %P63 の集まり
%P0 . . . %P63	プレディケート (1 ビット) レジスタ 0 . . . 63
	分岐レジスタ
%RP (%B0)	リターン・ポインタ
%B1 . . . %B7	分岐レジスタ 1 . . . 7
	汎用整数レジスタ
%R0	汎用整数レジスタ 0
%GP (%R1)	グローバル・データ・ポインタ
%R2 . . . %R11	汎用整数レジスタ 2 . . . 11
%SP (%R12)	スタック・ポインタ
%TP (%R13)	スレッド・ポインタ
%R14 . . . %R24	汎用整数レジスタ 14 . . . 24
%AP (%R25)	引数情報
%R26 . . . %R127	汎用整数レジスタ 26 . . . 127
	出力レジスタ
%OUT0 . . . %OUT7	出力レジスタ，実行時別名 (たとえば，フレームが出力レジスタに割り当てられた場合，%OUT0 は，最初に割り当てられた出力レジスタ，たとえば%R38 に対応する)。

(次ページに続く)

表 4–2 (続き) Integrity レジスタのデバッガ・シンボル

シンボル	説明
	汎用レジスタ
%GRNAT0 および%GRNAT1	それぞれ 64 ビットの汎用レジスタの NAT (Not A Thing) コレクション・レジスタ。たとえば%GRNAT0<3,1,0>は，%R3 の NAT ビット。
	浮動小数点レジスタ
%F0 . . . %F127	浮動小数点レジスタ 0 . . . 127

Integrity プロセッサには次のことが該当します。

- プログラムで同じ名前のシンボルを定義していない場合は，パーセント記号 (%) の接頭辞を省略できます。
- 未割り当てのレジスタ，無効状態のレジスタ，または読み取り不可のレジスタには値を格納できません。例を次に示します。
  - %R38 ～%R127 (%R32 ～%R37 だけが割り当てられた場合)
  - %F0 (常に 0.0)
  - %F1 (常に 1.0)
  - %R0 (常に 0)
  - %SP
  - %P0 (常に 1)
  - %GRNAT0 および%GRNAT1
  - %PC 以外のすべての特殊レジスタ
  - 大半の制御レジスタとアプリケーション・レジスタ (下記を参照)
- 通常のユーザ・モードのデバッグと SCD では，次のレジスタにも値を格納できます。
  - 例外フレーム用制御レジスタ%IPSR，%ISR，%IIP，%IFA，%ITIR，%IIPA，%IFS，%IIM，%IHA
  - アプリケーション・レジスタ%RSC，%CCV
- SCD では，次のレジスタにも値を格納できます。
  - アプリケーション・レジスタ%KR0 ～%KR7
  - 制御レジスタ%DCR，%ITM，%IVA，%PTA，%LID，%TPR，%IRR0 ～%IRR3，%ITV，%PMV，%CMCV，%LRR0，%LRR1
- ベクタ・レジスタはありません。
- 一部のレジスタ読み取りは，自動的に書式化されます。第 4.4.1 項で説明しているように，この書式は変更できます (たとえば， EXAMINE/QUAD/HEX %FPSR)。

- 浮動小数点ステータス・レジスタ (%FPSR) についての詳細は，『*Intel IA-64* **アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル第** *1* **巻**』を参照してください。例を次に示します。

```
DBG> ex %fpsr
LOOPER\main\%FPSR:
        I U O Z D V TD RC PC WRE FTZ
    SF3 0 0 0 0 0 0  1  0  3   0   0
    SF2 0 0 0 0 0 0  1  0  3   0   0
    SF1 0 0 0 0 0 0  1  0  3   1   0
    SF0 0 0 0 0 0 0  0  0  3   0   0
    TRAPS ID UD OD ZD DD VD
          1  1  1  1  1  1
DBG>
```

  この書式付けを，任意の位置で強制することもできます (EXAMINE/FPSR を参照)。

- 以前のファンクション状態 (%PFS) レジスタ，現在のフレーム・マーカ (%CFM) レジスタ，割り込みファンクション状態 (%IFS) レジスタ，前々回のファンクション状態 (%NEXT_PFS) レジスタについての詳細は，『*Intel IA-64* **アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル第** *1* **巻**』を参照してください。例を次に示します。

```
DBG> ex %pfs
LOOPER\main\%PFS:
    PPL PEC SOF SOL SOR RRB_GR RRB_FR RRB_PR
      3   0  29  21   0      0      0      0
DBG> ex %cfm
LOOPER\main\%CFM:
    SOF SOL SOR RRB_GR RRB_FR RRB_PR
      6   5   0      0      0      0
DBG> ex %ifs
LOOPER\main\%IFS:
    SOF SOL SOR RRB_GR RRB_FR RRB_PR
      6   5   0      0      0      0
DBG> ex %next_pfs
LOOPER\main\%NEXT_PFS:
    PPL PEC SOF SOL SOR RRB_GR RRB_FR RRB_PR
      3   0   6   5   0      0      0      0
DBG>
```

  EXAMINE/PFS と EXAMINE/CFM も参照してください。

- プロセッサ・ステータス・レジスタ (%PSR) についての詳細は，『*Intel IA-64* **アーキテクチャ・ソフトウェア・デベロッパーズ・マニュアル第** *2* **巻**』を参照してください。例を次に示します。

```
DBG> ex %psr
LOOPER\main\%PSR:
    IA BN ED RI SS DD DA ID IT MC IS CPL RT TB LP DB SI DI PP SP DFH DFL
     0  1  0  0  0  0  0  0  1  0  0   3  1  0  0  0  0  1  0  0   0   0
    DT PK  I IC MFH MFL AC UP BE
     1  0  1  1   1   1  0  0  0
DBG>
```

EXAMINE/PSR も参照してください。

- デバッガは，%GRNAT0 レジスタ，%GRANT1 レジスタ，および%PR レジスタに対して，省略時はビット・ベクタ・フォーマットを使用します。例を次に示します。

```
DBG> ex %grnat0,%pr
LOOPER\main\%GRNAT0:
11111111 11111111 11111111 11000000 00000000 00000000 00000000 00000000
LOOPER\main\%PR:
00000000 00000000 00000000 00000000 11111111 01010110 10010110 10100011
DBG>
```

- デバッガは，%p0 ～%p63 レジスタに対して，省略時は 1 ビット・フォーマットを使用します。例を次に示します。

```
DBG> ex %p6,%p7
LOOPER\main\%P6:        0
LOOPER\main\%P7:        1
DBG>
```

4.5 検査と格納を行う場合の型の指定

ここまでの節では，シンボリック名を持ち，したがってコンパイラ生成型に対応しているプログラム記憶位置を EXAMINE コマンドと DEPOSIT コマンドで使用する方法を説明しました。

第 4.5.1 項では，シンボリック名を持たないプログラム記憶位置用にデバッガがデータを編集 (型指定) する方法と，それらの記憶位置用に型を制御する方法を説明します。

第 4.5.2 項では，プログラム記憶位置に対応した型を上書きする方法を，シンボリック名を持つ記憶位置も含めて説明します。

4.5.1 シンボリック名を持たないプログラム記憶位置に対する型の定義

シンボリック名を持たず，したがってコンパイラ生成型に対応付けられていないプログラム記憶位置は，省略時の設定ではロングワード整数型を持ちます。この省略時の型を使用してそれらの記憶位置を検査し，それらの記憶位置へ値を格納する方法については，第 4.1.5 項を参照してください。

SET TYPE コマンドを使用すると，省略時の型を変更できます。これは，別の型の記憶位置の内容を検査および表示したい場合や，特定の型の値を別の型に対応付けられた記憶位置へ格納したい場合に役立ちます。表 4–3 は， SET TYPE コマンドの型のキーワードの一覧です。

表 4–3 SET TYPE キーワード

ASCIC	D_FLOAT		PACKED
ASCID	DATE_TIME	INSTRUCTION	QUADWORD
ASCII:*n*	EXTENDED_FLOAT[1]	LONG_FLOAT[1]	S_FLOAT[1]
ASCIW	F_LOAT	LONG_LONG_FLOAT[1]	T_FLOAT[1]
ASCIZ	FLOAT	LONGWORD	TYPE=(*type-expression*)
BYTE	G_FLOAT	OCTAWORD	WORD
			X_FLOAT[1]

[1]Integrity および Alpha 固有

たとえば，次のコマンドはシンボリック名を持たない記憶位置の型をバイト整数型，G 浮動小数点数型， 6 バイトの ASCII データからなる ASCII 型にそれぞれ設定します。一連の SET TYPE コマンドは型を再設定します。

```
DBG> SET TYPE BYTE
DBG> SET TYPE G_FLOAT
DBG> SET TYPE ASCII:6
```

SET TYPE コマンドは，/OVERRIDE 修飾子を指定せずに使用した場合，シンボリック名を持つプログラム記憶位置 (コンパイラ生成型に対応付けられた記憶位置) の型には影響を及ぼさないので注意してください。

SHOW TYPE コマンドは，シンボリック名を持たない記憶位置の現在の型を表示します。そのような記憶位置の省略時の型を復元するには， SET TYPE LONGWORD コマンドを入力します。

4.5.2 現在の型の上書き

SET TYPE/OVERRIDE コマンドを使用すれば，任意のプログラム記憶位置に対応した型を上書きでき，その結果どのようなコンパイラ生成型でも変更できます。たとえば，次のコマンドの実行後に EXAMINE コマンドを無修飾で実行した場合，指定した記憶位置の最初のバイトの内容だけが表示され，その内容はバイト整数データとして解釈されます。無修飾の DEPOSIT コマンドを実行した場合は，指定した記憶位置の最初のバイトだけが変更され，格納するデータはバイト整数データとして編集されます。

```
DBG> SET TYPE/OVERRIDE BYTE
```

SET TYPE/OVERRIDE コマンドに対して指定できる型キーワードについては，表 4–3 を参照してください。

現在の上書き型を表示するには，SHOW TYPE/OVERRIDE コマンドを入力します。現在の上書き型を取り消して，シンボリック名を持つ記憶位置の通常の解釈へ戻すには，CANCEL TYPE/OVERRIDE コマンドを入力します。

EXAMINE コマンドと DEPOSIT コマンドには，EXAMINE コマンドまたは DEPOSIT コマンドの実行中に，プログラムの記憶位置に現在割り当てられている型を無効にするための修飾子があります。これらの修飾子は，コンパイラで生成された型だけでなく，SET TYPE または SET TYPE/OVERRIDE コマンドも無効にします。各コマンドで使用できる型修飾子については，DEPOSIT コマンドと EXAMINE コマンドを参照してください。

型修飾子を指定して EXAMINE コマンドを使用した場合，アドレス式で指定した要素がその型で表示されます。次に例を示します。

```
DBG> EXAMINE/BYTE .           ! 型はバイト整数である。
MOD3\%LINE 15 :  -48
DBG> EXAMINE/WORD .           ! 型はワード整数である。
MOD3\%LINE 15 :  464
DBG> EXAMINE/LONG .           ! 型はロングワード整数である。
MOD3\%LINE 15 :  749404624
DBG> EXAMINE/QUAD .           ! 型はクォドワード整数である。
MOD3%LINE 15 :  +0130653502894178768
DBG> EXAMINE/FLOAT .          ! 型は F 浮動小数点数である。
MOD3%LINE 15 :   1.9117807E-38
DBG> EXAMINE/G_FLOAT .        ! 型は G 浮動小数点数である。
MOD3%LINE 15 :   1.509506018605227E-300
DBG> EXAMINE/ASCII .          ! 型は ASCII 文字列である。
MOD3\%LINE 15 :  ".."
DBG>
```

型修飾子を指定して DEPOSIT コマンドを使用した場合，アドレス式で指定した記憶位置へその型の値が格納され，そのアドレス式に対応する型が上書きされます。

残りの項では，型修飾子と SET TYPE コマンドを使用して整数型，文字列型，およびユーザ宣言型を指定する例を示します。

4.5.2.1 整数型

次の例は，整数型修飾子 (/BYTE，/WORD，/LONGWORD) を指定した EXAMINE コマンドと DEPOSIT コマンドの使用例です。これらの型修飾子を使用すれば，特定の整数型の値を任意のプログラム記憶位置へ格納することができます。

```
DBG> SHOW TYPE           ! コンパイラ生成型を持たない
type:  long integer      ! 記憶位置の型を表示する。
DBG> EVALU/ADDR .        ! 現在の記憶位置は 724 である。
724
DBG> DEPO/BYTE . = 1     ! 値 1 をアドレス 724 の 1 バイトの
                         ! メモリへ格納する。
DBG> EXAM .              ! 省略時の設定では 4 バイトが検査される。
724:  1280461057
DBG> EXAM/BYTE .         ! 1 バイトだけを検査する。
724:  1
DBG> DEPO/WORD . = 2     ! 値 2 を現在の値の最初の
                         ! 2 バイト（ワード）へ格納する。
DBG> EXAM/WORD .         ! 現在の値の 1 ワードを検査する。
724:  2
DBG> DEPO/LONG 724 = 999 ! 値 999 をアドレス 724 から始まる 4 バイト
                         !(ロングワード）へ格納する。
DBG> EXAM/LONG 724       ! アドレス 724 から始まる 4 バイト
724:  999                ! (ロングワード）を検査する。
DBG>
```

4.5.2.2 ASCII 文字列型

次の例は，/ASCII:*n*型修飾子を指定した EXAMINE コマンドと DEPOSIT コマンドの使用例です。

この修飾子を DEPOSIT コマンドで使用すると，長さ*n*の ASCII 文字列を任意のプログラム記憶位置に格納することができます。この例では，記憶位置はシンボリック名 (I) を持っており，したがってコンパイラ生成型に対応付けられています。コマンドの形式は次のとおりです。

```
DEPOSIT/ASCII:n address-expression = "ASCII string of length n"
```

*n*の省略時の値は 4 バイトです。

```
DBG> DEPOSIT I = "abcde"    ! I はコンパイラ生成の整数型を持つ。
%DEBUG-W-INVNUMBER, invalid numeric string 'abcde'
                            ! したがって，I には文字列を格納できない。
DBG> DEP/ASCII:5 I = "abcde"! /ASCII 修飾子によって整数型を上書きし，
                            ! 5 バイトの ASCII データを格納
                            ! できるようにする。
DBG> EXAMINE .              ! I の値をコンパイラ生成の整数型で

MOD3\I:  1146048327         ! 表示する。
DBG> EXAM/ASCII:5 .         ! I の値を 5 バイトの ASCII 文字列として
MOD3\I:  "abcde"            ! 表示する。
DBG>
```

複数の DEPOSIT/ASCII コマンドを入力する場合は， SET TYPE/OVERRIDE コマンドを使用して上書き ASCII 型を設定できます。その場合，それ以後の EXAMINE コマンドと DEPOSIT コマンドは/ASCII 修飾子を指定したのと同じ効果を持ちます。次に例を示します。

```
DBG> SET TYPE/OVER ASCII:5! ASCII:5 を上書き型として設定する。
DBG> DEPOSIT I = "abcde"  !     I に 5 バイトの文字列を格納できるようになる。
DBG> EXAMINE I            !     I の値を 5 バイトの
MOD3\I:  "abcde"     ! ASCII 文字列として表示する。
DBG> CANCEL TYPE/OVERRIDE ! ASCII 上書き型を取り消す。
DBG> EXAMINE I            !     I をコンパイラ生成型で表示する。
MOD3\I:  1146048327
DBG>
```

4.5.2.3 ユーザ宣言型

次の例は，/TYPE=(*name*) 修飾子を指定した EXAMINE コマンドと DEPOSIT コマンドの使用例です。この修飾子を使用すれば，検査または格納を行う際にユーザ宣言上書き型を指定できます。

たとえば， Pascal プログラムに次のコードが入っているとします。このコードは 3 つの値 RED， GREEN，および BLUE を持つ列挙型 COLOR を宣言します。

```
   .
   .
   .
TYPE
    COLOR = (RED,GREEN,BLUE);
   .
   .
   .
```

デバッグ・セッションの間，次のように SHOW SYMBOL/TYPE コマンドはデバッガが認識しているとおりに型 COLOR を示します。

```
DBG> SHOW SYMBOL/TYPE COLOR
data MOD3\COLOR
    enumeration type (COLOR, 3 elements), size: 1 byte
DBG>
```

次のコマンドはアドレス 1000 の値を表示します。この値はシンボリック名に対応付けられていません。したがって，省略時の設定では値 0 がロングワード整数型として表示されます。

```
DBG> EXAMINE 1000
1000:   0
DBG>
```

次のコマンドはアドレス 1000 の値を型 COLOR で表示します。前の SHOW SYMBOL/TYPE コマンドで，それぞれの列挙要素が 1 バイトに格納されていることが示されています。したがって，デバッガはアドレス 1000 にあるロングワード整数値 0 の最初のバイトを，それに相当する列挙値 RED (3 つの列挙値の最初のもの) に変換します。

```
DBG> EXAMINE/TYPE=(COLOR) 1000
1000:   RED
DBG>
```

次の DEPOSIT コマンドは，上書き型 COLOR を持つアドレス 1000 に値 GREEN を格納します。 EXAMINE コマンドは，アドレス 1000 の値を省略時の型であるロングワード整数で表示します。

```
DBG> DEPOSIT/TYPE=(COLOR) 1000 = GREEN
DBG> EXAMINE 1000
1000:   1
DBG>
```

次の SET TYPE コマンドは，シンボリック名を持たないアドレス 1000 のような記憶位置に対して型 COLOR を設定します。 EXAMINE コマンドは， 1000 にある値を型 COLOR で表示するようになります。

```
DBG> SET TYPE TYPE=(COLOR)
DBG> EXAMINE 1000
1000:   GREEN
DBG>
```

5
プログラム内シンボルへのアクセス制御

シンボリック・デバッグでは，ソース・コード内に現れるとおりに，変数名，ルーチン名などを指定できます。ユーザは，プログラム記憶位置を参照する場合，数値メモリ・アドレスやレジスタを使用する必要はありません。

さらにユーザは，プログラムとそのソース言語に適切なコンテキストでシンボルを使用できます。デバッガは，要素の有効範囲と可視性，データ型，式などについて，使用している言語の規則をサポートします。

プログラムに対応するすべてのシンボルにアクセスするためには，/DEBUG コマンド修飾子を使用してそのプログラムをコンパイルおよびリンクしなければなりません。

これらの条件のもとでは，シンボル情報がソース・プログラムからデバッガに渡され処理される方法は，ほとんどの場合ユーザには見えません。しかし，何らかの対処が必要な場合もあります。

たとえば， COUNTER という名前のルーチンにブレークポイントを設定しようとすると，次の診断メッセージが表示されます。

```
DBG> SET BREAK COUNTER
%DEBUG-E-NOSYMBOL, symbol 'COUNTER' is not in the symbol table
DBG>
```

このような場合，第 5.2 節で説明するように， COUTNTER が定義されているモジュールを設定しなければなりません。

シンボル X が， 2 つ以上のモジュール，ルーチン，または他のプログラム単位内で定義 (宣言) されている場合は，次のメッセージが表示されます。

```
DBG> EXAMINE X
%DEBUG-E-NOUNIQUE, symbol 'X' is not unique
DBG>
```

このような場合，第 5.3 節で説明するように，シンボルのパス名を指定することによってシンボルのあいまいさを解消しなければなりません。

本章では，プログラム内シンボルのアクセスに関連する，上記およびその他の状況に対処する方法について説明します。

本章で説明するシンボル (通常は，アドレス式) は，ソース・プログラム内に存在するものだけです。

- ソース・コード内で宣言された要素 (変数，ルーチン，ラベル，配列要素，およびレコード構成要素など) の名前。
- ユーザ・プログラムとリンクされた共用可能イメージおよびモジュール (コンパイル単位) の名前。
- デバッガがソース・コードを識別するために使用する要素。たとえば，ソース・ファイルの仕様，およびリスト・ファイル内やソース・コード表示時のソース行番号など。

次のシンボルについては別の章で説明します。

- デバッグ・セッション中にユーザが DEFINE コマンドを使用して作成したシンボルについては，第 13.4 節で説明します。
- ピリオド(.)や%PC などの組み込みシンボルについては，本書を通して適切な箇所で説明します。また，付録 B にその定義があります。

また，シンボリック・アドレス式に対応するメモリ・アドレスやレジスタ名を取得したり，逆にプログラム記憶位置をシンボル化する方法についての詳しい説明は，第 4.1.11 項を参照してください。

注意

プログラムがコンパイル時に最適化されている場合，プログラム内の特定の変数がコンパイラによって除去されることがあります。このような変数を参照しようとすると，デバッガは警告を発行します (第 1.2 節および第 14.1 節を参照)。

非静的変数 (スタックローカル変数またはレジスタ変数) を参照する前には，その変数を定義しているルーチンが呼び出しスタック上でアクティブでなければなりません。すなわち，プログラムの実行が，非静的変数を定義しているルーチン内で停止していなければなりません (第 3.4.3 項を参照)。

5.1 コンパイル時およびリンク時のシンボル情報制御

シンボリック・デバッグの機能を最大限に利用するためには，第 1.2 節で説明したように，/DEBUG 修飾子を使用してプログラムをコンパイルおよびリンクする必要があります。

次の各項では，コンパイル時およびリンク時に，シンボル情報がどのように作成され，デバッガに渡されるのかを説明します。

5.1.1 コンパイル

/DEBUG 修飾子を使用してソース・ファイルをコンパイルすると，コンパイラは，デバッグ・シンボル・テーブル用のシンボル・レコード (DST レコード) を作成し，作成中のオブジェクト・モジュールにそれらを取り込みます。

DST レコードは，シンボル名だけでなく，次に示すようにその使用法に関連するすべての情報を提供します。

- データ型，範囲，制約，および変数の有効範囲
- 関数およびプロシージャのパラメータ名とパラメータ・タイプ
- ソース行相関関係レコード (ソース行を行番号とソース・ファイルに対応づける)

ほとんどのコンパイラでは，異なるオプションを/DEBUG 修飾子とともに指定すると，オブジェクト・モジュールに取り込む DST 情報の量を調整することができます。表 5–1 に，ほとんどのコンパイラのオプションを示します。詳しい説明は，コンパイラ付属のドキュメントを参照してください。

表 5–1 DST シンボル情報用のコンパイラ・オプション

コンパイラ・コマンド修飾子	オブジェクト・モジュール内の DST 情報
/DEBUG[1]	すべての情報
/DEBUG=TRACEBACK[2]	トレースバック情報のみ (モジュール名，ルーチン名，行番号)
/NODEBUG[3]	情報なし

[1] /DEBUG，/DEBUG=ALL，および/DEBUG=(SYMBOLS,TRACEBACK) は等価です。
[2] /DEBUG=TRACEBACK および DEBUG=(NOSYMBOLS,TRACEBACK) は等価です。
[3] /NODEBUG，/DEBUG=NONE，および/DEBUG=(NOSYMBOLS,NOTRACEBAC K) は等価です。

ほとんどのコンパイラの省略時の設定は TRACEBACK オプションです。すなわち，/DEBUG 修飾子を省略すると，ほとんどのコンパイラは，/DEBUG=TRACEBACK が指定されたものとみなします。 TRACEBACK オプションを使用すると，実行時エラー発生時にシンボリック・トレースバックを出力できるように，トレースバック条件ハンドラがメモリ・アドレスをルーチン名と行番号に変換します。次に例を示します。

```
$ RUN FORMS
   .
   .
   .
%PAS-F-ERRACCFIL, error in accessing file PAS$OUTPUT
%PAS-F-ERROPECRE, error opening/creating file
%RMS-F-FNM, error in file name
%TRACE-F-TRACEBACK, symbolic stack dump follows

module name     routine name     line      rel PC      abs PC
```

```
PAS$IO_BASIC    _PAS$CODE                00000192    00001CED
PAS$IO_BASIC    _PAS$CODE                0000054D    000020A8
PAS$IO_BASIC    _PAS$CODE                0000028B    00001DE6
FORMS           FORMS             59     00000020    000005A1
$
```

トレースバック情報は，デバッガの SHOW CALLS コマンドによっても使用されます。

5.1.2 ローカル・シンボルとグローバル・シンボル

DST レコードは，プログラム内で定義されているすべてのシンボルに関する情報を含んでいます。これらのシンボルはローカル・シンボルかグローバル・シンボルのどちらかです。

通常，**ローカル・シンボル**とは，それが定義されているモジュール内でのみ参照されるシンボルを意味します。一方**グローバル・シンボル**とは，あるモジュール内で定義されていますが，別のモジュールから参照されるシンボルを意味します。ルーチン名，プロシージャ・エントリ・ポイント，またはグローバル・データ名などがグローバル・シンボルです。

ある共用可能イメージ内で定義され，別のイメージから参照されるグローバル・シンボル (プログラムの実行可能メイン・イメージなど) を**ユニバーサル・シンボル**と呼びます。共用可能イメージを作成する場合，リンク時にすべてのユニバーサル・シンボルを明示的に定義しなければなりません。ユニバーサル・シンボルと共用可能イメージについての詳しい説明は第 5.4 節を参照してください。

通常，コンパイラがローカル・シンボルへの参照を解決し，リンカがグローバル・シンボルへの参照を解決します。

ローカル・シンボルとグローバル・シンボルとの違いについては，シンボル検索および共用可能イメージとユニバーサル・シンボルとの関連で，本章の各箇所で説明します。

5.1.3 リンク

ユーザが LINK/DEBUG コマンドを入力し，オブジェクト・モジュールをリンクして実行可能なイメージを作成する場合，リンカはデバッグに影響を与えるいくつかの機能を実行します。

- リンク中のオブジェクト・モジュールに含まれている DST レコードから，デバッグ・シンボル・テーブル (DST) を作成する。 DST は，デバッグ・セッション中のシンボル情報のおもなソースである。

- グローバル・シンボルへの参照を解決して，グローバル・シンボル・テーブル (GST) を作成する。 GST は，すでに DST 内に含まれているグローバル・シンボル情報のいくつかをコピーしたものである。 GST は，デバッグ時の特定の条件下ではシンボル検索に使用される。
- DST と GST を実行可能なイメージ内に取り込む。
- 実行可能なイメージ内にフラグを設定し，ユーザが DCL コマンド RUN を入力したときに，イメージ・アクティベータがデバッガに制御を渡すようにする (第 1.2 節を参照)。

第 5.4 節では，共用可能イメージをデバッグ用にリンクする方法を，ユニバーサル・シンボル (共用可能イメージ内に定義され，別のイメージから参照されるグローバル・シンボル) の定義方法といっしょに説明します。

表 5–2 には，デバッガに渡される DST 情報および GST 情報のレベルをコンパイラまたは LINK コマンド・オプション別にまとめてあります。コンパイラ・コマンド修飾子は，リンカに渡される DST 情報のレベルを制御します。 LINK コマンド修飾子は，デバッガに渡される DST 情報と GST 情報の量だけでなく，プログラムをデバッガの制御下に置くことができるかどうかも制御します (第 1.2 節を参照)。

表 5–2 DST および GST のシンボル情報に与えるコンパイラとリンカの影響

コンパイラ・コマンド修飾子[1]	オブジェクト・モジュール内の DST データ	LINK コマンド修飾子[2]	デバッガに渡される DST データ	デバッガに渡される GST データ[3]
/DEBUG	すべての情報	/DEBUG	すべての情報	すべての情報
/DEBUG=TRACE	トレースバック情報のみ	/DEBUG	トレースバック情報のみ	すべての情報
/NODEBUG	情報なし	/DEBUG	情報なし	すべての情報
/DEBUG	すべての情報	/DSF[4]	すべての情報	すべての情報[5]
/DEBUG=TRACE	トレースバック情報のみ	/DSF[4]	トレースバック情報のみ	すべての情報[5]
/NODEBUG	情報なし	/DSF[4]	情報なし	すべての情報[5]
/DEBUG	すべての情報	/TRACE[6]	トレースバック情報のみ	すべての情報
/DEBUG=TRACE	トレースバック情報のみ	/TRACE	トレースバック情報のみ	すべての情報
/NODEBUG	情報なし	/TRACE	情報なし	すべての情報

[1]詳しい説明は，表 5–1 を参照してください。

[2]共用可能イメージを作成する場合は，/SHAREABLE 修飾子も指定する必要があります (第 5.4 節を参照)。

[3]GST データには，リンク時に解決されるグローバル・シンボル情報が含まれています。実行可能なイメージの GST データには，グローバル・ルーチンおよびグローバル定数の名前と値が含まれます。共用可能イメージの GST データには，ユニバーサル・シンボルが含まれます (第 5.1.2 項および第 5.4 節を参照)。

[4]Alpha のみ。

[5]DBG$IMAGE_DSF_PATH は.DSF ファイルがあるディレクトリを指していなければなりません。

[6]LINK/TRACEBACK と LINK/NODEBUG は等価です。これは LINK コマンドの省略時の設定です。

(次ページに続く)

表 5–2 (続き) DST および GST のシンボル情報に与えるコンパイラとリンカの影響

コンパイラ・コマンド修飾子[1]	オブジェクト・モジュール内の DST データ	LINK コマンド修飾子[2]	デバッガに渡される DST データ	デバッガに渡される GST データ[3]
/DEBUG	すべての情報	/NOTRACE[7]		

[1]詳しい説明は，表 5–1 を参照してください。

[2]共用可能イメージを作成する場合は，/SHAREABLE 修飾子も指定する必要があります (第 5.4 節を参照)。

[3]GST データには，リンク時に解決されるグローバル・シンボル情報が含まれています。実行可能なイメージの GST データには，グローバル・ルーチンおよびグローバル定数の名前と値が含まれます。共用可能イメージの GST データには，ユニバーサル・シンボルが含まれます (第 5.1.2 項および第 5.4 節を参照)。

[7]RUN/DEBUG コマンドを使用すれば，デバッガを起動できますが， LINK/NOTRACEBACK コマンドを使用してリンクした場合は，シンボリック・デバッグは実行できません。

コンパイラ・コマンドに/NODEBUG 修飾子を指定し，その後イメージをリンクし実行すると，デバッガは，プログラムがデバッガの制御下に置かれたときに次のようなメッセージを発行します。

```
%DEBUG-I-NOLOCALS, image does not contain local symbols
```

上記のメッセージは，/TRACEBACK 修飾子または/DEBUG 修飾子のどちらを使用してリンクした場合にも発行され，リンクされたイメージの DST が作成されなかったことを意味します。この場合， GST 内に含まれるグローバル・シンボルだけが使用可能となります。

LINK コマンドに/DEBUG 修飾子を指定しないと，デバッガは，プログラムがデバッガの制御下に置かれたときに次のようなメッセージを発行します。

```
%DEBUG-I-NOGLOBALS, some or all global symbols not accessible
```

上記のメッセージは，デバッグ・セッション中に使用可能なグローバル・シンボル情報は， DST 内に格納されています。

これらの概念については，次の各節で説明します。特に，共用可能イメージのデバッグについての詳しい説明は，第 5.4 節を参照してください。

5.1.4 デバッグ済みイメージ内のシンボル情報制御

シンボル・レコードは，実行可能なイメージ内で領域を占有します。プログラムのデバッグ終了後，実行可能なイメージを小さくするために，/DEBUG 修飾子を使用しないで再度リンクする場合があります。この場合， DST (トレースバック・データのみ) および GST を含むイメージが作成されます。

LINK/NOTRACEBACK コマンドを使用すると，デバッグの終了後，イメージ内容をユーザによる変更から保護することができます。このコマンドは，特権を使用してインストールする必要のあるイメージに使用します。(『OpenVMS システム管理者マ

ニュアル』および『OpenVMS **システム管理ユーティリティ・リファレンス・マニュアル**』を参照してください。) /NOTRACEBACK 修飾子を LINK コマンドと使用すると，トレースバック・データを含むシンボル情報はイメージに渡されません。

5.1.5 個別のシンボル・ファイルの作成 (Alpha のみ)

Alpha システムでは，/DSF 修飾子を使用してプログラムをリンクすることにより，シンボル情報を格納した個別のファイルを作成できます。省略時の設定では，シンボル・ファイルは， LINK ユーティリティで作成される実行可能ファイルと同じファイル名であり，ファイル・タイプは.DSF です。次の例を参照してください。

```
$ CC/DEBUG/NOOPTIMIZE TESTPROGRAM.C
$ LINK/DSF TESTPROGRAM
$ DEFINE DBG$IMAGE_DSF_PATH SYS$DISK:[]
$ DEBUG/KEEP TESTPROGRAM
```

この例では，次の処理が実行されます。

1. TESTPROGRAM.C をコンパイルする。
2. TESTPROGRAM.EXE と TESTPROGRAM.DSF を作成する。
3. 論理名 DBG$IMAGE_DSF_PATH をカレント・ディレクトリとして定義する。
4. 保持デバッガでデバッガを起動する。

このプローシージャを使用すると，これまでより小さい実行可能ファイルを作成し，しかもデバッグのためにグローバル・シンボル情報を使用できます。インストール済み常駐ファイルなど，特定のアプリケーションでは，実行可能ファイルにシンボル・テーブルを格納できません。さらに，.DSF ファイルを使用すると，シンボル・テーブルを含まない実行可能ファイルを顧客に配布することができ，しかも将来デバッグが必要になったときのために，別に.DSF ファイルを保存しておくことができます。

注意

デバッグを簡単にするために，プログラムをコンパイルするときは，/NOOPTIMIZE 修飾子を使用してください (可能な場合)。最適化されたコードのデバッグについては，第 14.1 節を参照してください。

別にシンボル (.DSF) ファイルが作成された実行可能ファイルをデバッグするには，次の条件を満たさなければなりません。

- .DSF ファイルの名前は，デバッグする.EXE ファイルの名前と同じでなければならない。
- .DSF ファイルを格納するディレクトリを指すように， DBG$IMAGE_DSF_PATH を定義しなければならない。

/DSF 修飾子の使い方については，『OpenVMS Linker Utility Manual』を参照してください。

5.2 モジュールの設定と取り消し

デバッガが，指定されたシンボル (たとえば，変数名 X) を検索できないため，次のようなメッセージを発行した場合，ユーザはモジュールを設定する必要があります。

```
DBG> EXAMINE X
%DEBUG-E-NOSYMBOL, symbol 'X' is not in the symbol table
DBG>
```

この節では，モジュール設定と，モジュールの設定または取り消しが必要となる条件について説明します。モジュールの設定には SET MODULE コマンド，取り消しには CANCEL MODULE コマンドをそれぞれ使用します。

/DEBUG コマンド修飾子を使用して，プログラムをコンパイルおよびリンクすると，第 5.1 節で説明したように，全シンボル情報がプログラムのソース・コードから実行可能なイメージに渡されます。

シンボル情報は，実行可能なイメージのデバッグ・シンボル・テーブル (DST) とグローバル・シンボル・テーブル (GST) に含まれています。 DST には，ローカル・シンボルおよびグローバル・シンボルに関する詳細な情報が含まれます。 GST は， DST に含まれるグローバル・シンボル情報をコピーしたものです。

デバッガは，シンボル検索を容易にするために， DST と GST から実行時シンボル・テーブル (RST) にシンボル情報をロードします。 RST は，効率的なシンボル検索のために構築されています。シンボル情報が RST 内に存在しない場合，デバッガはそのシンボルを認識しないか，または正確に解釈しません。

RST はメモリ領域を占有するため，デバッガは，プログラムの実行中に参照されると思われるシンボルを予想しながら，動的にシンボルをロードします。特定のモジュールのすべてのシンボル情報が，一度に RST テーブル内にロードされるため，このローディング・プロセスを**モジュール設定**と呼びます。

グローバル・シンボルは，デバッグ・セッション中は常にアクセス可能でなければならないので，プログラムがデバッガの制御下に置かれたとき，すべての GST レコードは， RST 内にロードされます。また，デバッガはメイン・プログラム (イメージ転送アドレスによって指定されるルーチンであり，デバッグ・セッションの開始時に実行が停止する位置) を含むモジュールを設定します。その結果，すべてのグローバル・シンボルと，メイン・プログラムから見えなければならない任意のローカル・シンボルにアクセスできるようになります。

その後，プログラムの実行が中断するたびに，実行が中断しているルーチンを含むモジュールが設定されます。 Ada プログラムでは，デバッガは， with 句またはサブユニット関係によって関連づけられた任意のモジュールも設定します (デバッガのオンライン・ヘルプを参照。Help Language_Support_Adaと入力します)。モジュールの設定によって，ユーザはグローバル・シンボルに加えてそのプログラム記憶位置で見えなければならないシンボルを参照できるようになります。この省略時の動作モードを動的モードと呼びます。モジュールが動的に設定されると，デバッガは次のようなメッセージを発行します。

```
%DEBUG-I-DYNMODSET, setting module MOD4
```

モジュール内に定義されているシンボルで，まだ設定されていないものを参照しようとすると，デバッガは，そのシンボルが RST に設定されていないことをユーザに警告します。このような場合は， SET MODULE コマンドを使用して，そのシンボルを含むモジュールを明示的に設定しなければなりません。次に例を示します。

```
DBG> EXAMINE X
%DEBUG-E-NOSYMBOL, symbol 'X' is not in the symbol table
DBG> SET MODULE MOD3
DBG> EXAMINE X
MOD3\ROUT2\X: 26
DBG>
```

SHOW MODULE コマンドは，プログラム内のモジュールの一覧を表示し，設定されているモジュールを識別します。

モジュールが設定されると，デバッガは， RST に必要なメモリを自動的に割り当てます。したがって，設定されるモジュール数が増えると，デバッガの処理速度が低下します。性能の低下が問題になる場合は， CANCEL MODULE コマンドを使用して設定モジュール数を減らし，自動的にメモリ領域を解放します。あるいは SET MODE NODYNAMIC コマンドを入力して，動的モードを無効にすることもできます。動的モードが無効の場合，モジュールはデバッグ時に自動的には設定されません。動的モードになっているかどうかを判定するには， SHOW MODE コマンドを使用します。

Ada プログラム固有のモジュール設定についての詳しい説明は，デバッガのオンライン・ヘルプを参照してください (Help Language_Support_Adaと入力します)。

第 5.4 節では，共用可能イメージをデバッグするときのイメージとモジュールの設定方法について説明します。

5.3 シンボルのあいまいさの解消

シンボルのあいまいさは，シンボル (たとえば，変数名 X) が 2 つ以上のルーチンまたは他のプログラム単位内で定義されているときに起こります。

ほとんどの場合，デバッガはシンボルのあいまいさを自動的に解消します。それには，第 5.3.1 項で説明したように，まず現在設定されている言語の有効範囲および可視性の規則が適用され，次に呼び出しスタック上の呼び出しルーチンの順序に従います。

ただし，多重定義されているシンボルを指定すると，デバッガは次のように応答する場合もあります。

- ユーザの意図する特定のシンボル宣言を確定できない。次に例を示す。

```
DBG> EXAMINE X
%DEBUG-W-NOUNIQUE, symbol 'X' is not unique
DBG>
```

- ユーザが意図している宣言ではなく，現在の有効範囲内で見える宣言を参照する。

このような問題を解決するには，デバッガが，特定のシンボル宣言を検索する範囲をユーザが指定しなければなりません。次の例では，パス名 COUNTER\X によって，変数 X の特定の宣言を一意に指定しています。

```
DBG> EXAMINE COUNTER\X
COUNTER\X: 14
DBG>
```

次の各項では，有効範囲の概念とシンボルのあいまいさを解消する方法について説明します。

5.3.1 シンボル検索規則

ここでは，デバッガのシンボル検索方法について説明します。デバッガは，プログラミング言語の有効範囲と可視性の規則，およびデバッガ自身の規則を使用して，発生する可能性の最も高いシンボルのあいまいさを解消します。第 5.3.2 項および第 5.3.3 項では，ユーザが必要な場合に利用できる補助的な方法について説明します。

デバッガ・コマンド内にシンボルを指定するには，パス名または正確なシンボルを使用します。

パス名を使用すると，デバッガは，パス名接頭識別子によって示される有効範囲内でシンボルを検索します (第 5.3.2 項を参照)。

パス名接頭識別子を指定しないと，デバッガは，省略時の設定によって，次の段落で説明する方法を使用して実行時シンボル・テーブル (RST) を検索します。この省略時の動作は，第 5.3.3 項で説明するように， SET SCOPE コマンドを使用して変更できます。

最初に，デバッガは，現在設定されている言語の有効範囲と可視性の規則に従って，PC **範囲** (有効範囲 0 とも呼ばれる) 内でシンボルを検索します。これは，デバッガは通常，最初に現在の PC 値 (現在実行が停止している位置) を含むブロック内またはルーチン内を検索することを意味します。シンボルが見つからない場合は，すぐ外側のネスティング・レベルのプログラム単位を検索し，さらにその外側のプログラム単位へと検索を続けます。この方法は言語に依存しますが，多重定義されているシンボルの中から正しい宣言が選択されることを保証します。

しかし，ユーザは，言語によって定義された PC 範囲内で見つかるシンボルだけではなく，プログラム内の任意のシンボルも参照することができます。これは，任意の領域にブレークポイントを設定したり，任意の変数を検査したりするために必要です。PC 範囲内でシンボルが見つからない場合，デバッガは次の手順に従って検索を続けます。

PC 範囲の検索後，デバッガは，必要であれば呼び出し元ルーチンの有効範囲を検索し，見つからなければ，さらにその呼び出し元という要領で検索を続けます。完全な**有効範囲検索リスト**が，シンボリックに示されます (0, 1, 2, . . . , *n*)。ここで， 0 は PC 範囲を示し，*n*は呼び出しスタック上の呼び出し数を示します。各有効範囲 (呼び出しフレーム) 内では，デバッガはその言語の可視性規則を使用してシンボルを検索します。

呼び出しスタックを基にした，この検索リストを使用すれば，デバッガは多重定義されたシンボルを便利で予測可能な方法で示すことができます。

上記の検索方法を使用しても，シンボルが見つからない場合，デバッガは， RST の残りの部分，すなわち残りの設定モジュールとグローバル・シンボル・テーブル (GST) を検索します。ここまで検索が進むと，デバッガはシンボルのあいまいさを解消しようとしませんが，シンボルが 2 つ以上現れた場合には，次のようなメッセージを発行します。

```
%DEBUG-W-NOUNIQUE, symbol 'Y' is not unique
```

SET SCOPE コマンドを使用して，省略時のシンボル検索動作を変更した場合は，CANCEL SCOPE コマンドを使用して，省略時の動作を復元することができます。

5.3.2 シンボルを一意に指定するための SHOW SYMBOL コマンドとパス名の使用

シンボル参照が一意でないことをデバッガが示している場合は， SHOW SYMBOL コマンドを使用して，そのシンボルの可能なパス名をすべて表示し，そのうちの 1 つを指定することによってシンボルを一意に指定します。次に例を示します。

```
DBG> EXAMINE COUNT
%DEBUG-W-NOUNIQUE, symbol 'COUNT' is not unique

DBG> SHOW SYMBOL COUNT
data MOD7\ROUT3\BLOCK1\COUNT
data MOD4\ROUT2\COUNT
routine MOD2\ROUT1\ROUT3\COUNT

DBG> EXAMINE MOD4\ROUT2\COUNT
MOD4\ROUT2\COUNT: 12
DBG>
```

SHOW SYMBOL COUNT コマンドは， RST 内に存在するシンボル COUNT のすべての宣言を表示します。 COUNT の最初の 2 つの宣言は変数 (データ) です。最後の宣言はルーチン名です。各宣言は，そのパス名接頭識別子とともに表示され，その宣言に到達するためにたどらなければならないパス (検索範囲) を示しています。たとえば， MOD4\ ROUT2\ COUNT は，モジュール MOD4 内のルーチン ROUT2 に存在するシンボル COUNT の宣言を示しています。

パス名の形式は，次のとおりです。すなわち，パス名の最も左の要素は，シンボルを含むモジュールを示します。その後，右に移動するに従って，連続してネストされたルーチンおよびブロックが続き，最後にシンボルの特定の宣言 (一番右側の要素) に到達します。

シンボルは常にパス名とともに表示されますが，ユーザが，デバッガ・コマンドにパス名を指定する必要があるのは，あいまいさを解消する必要があるときだけです。

デバッガは，行番号を他のシンボルと同じように検索します。省略時の設定では，最初に，実行が停止したモジュール内を検索します。よく使用されるパス名の使用法に，任意のモジュール内の行番号を指定する場合があります。次に例を示します。

```
DBG> SET BREAK QUEUE_MANAGER\%LINE 26
```

次の例では SHOW SYMBOL コマンドが，グローバル・シンボルを 2 回示しています。これは，グローバル・シンボルが DST と GST の両方に含まれているためです。

```
DBG> SHOW SYMBOL X
data ALPHA\X                ! グローバル X
data ALPHA\BETA\X           ! ローカル X
data X (global)             ! ALPHA\X と同じ
DBG>
```

共用可能イメージの場合は，その中に含まれるグローバル・シンボルはユニバーサル・シンボルなので， SHOW SYMBOL コマンドは，ユニバーサル・シンボルを 2 回示します (第 5.1.2 項および第 5.4 節を参照)。

5.3.2.1 パス名の単純化

パス名は長くなることがあります。ユーザは，次の 3 つの方法でパス名の指定を単純化できます。

- パス名を簡略化する。
- パス名として簡単なシンボルを定義する。
- パス名を指定しなくてもよいように，新しい検索有効範囲を設定する。

パス名を簡略化するには，ネストされたプログラム単位を左から順番に削除して，一意にシンボルを指定するのに十分なパス名だけを残します。たとえば，第 5.3.2 項の最初の例では， ROUT3\COUNT は有効な簡略化されたパス名です。

パス名のシンボルを定義するには， DEFINE コマンドを使用します。次に例を示します。

```
DBG> DEFINE INTX = INT_STACK\CHECK\X
DBG> EXAMINE INTX
```

新しい検索有効範囲を設定するには， SET SCOPE コマンドを使用します。このコマンドについては，第 5.3.3 項を参照してください。

5.3.2.2 呼び出しスタック上ルーチン内のシンボルの指定

ユーザは，数値パス名を使用して， SHOW CALLS の表示によって示される呼び出しスタック上のルーチンに対応する有効範囲を指定できます。パス名接頭識別子の"0\ "は PC 範囲を示し，"1\ "は有効範囲 1 (呼び出し元の有効範囲) を示します。

たとえば，次のコマンドは，それぞれ有効範囲 0 および有効範囲 2 の中で見える， 2 つの異なる宣言 Y の現在値を表示します。

```
DBG> EXAMINE 0\Y
DBG> EXAMINE 2\Y
```

省略時の設定では， EXAMINE Y コマンドは， EXAMINE 0\Y を意味します。

第 5.3.3 項の SET SCOPE/CURRENT コマンドの説明を参照してください。このコマンドを使用すれば，省略時の有効範囲検索リストの参照を呼び出しスタックの相対位置に再設定できます。

5.3.2.3 グローバル・シンボルの指定

グローバル・シンボルを一意に指定するには，シンボルの接頭辞としてバックスラッシュ(\ )を使用します。たとえば，次のコマンドはグローバル・シンボル X の値を表示します。

```
DBG> EXAMINE \X
```

5.3.2.4 ルーチンの起動の指定

あるルーチンが再帰的に呼び出された場合，同じルーチンに対するいくつかの呼び出しを区別する必要があります。その結果，これらの呼び出しすべてに対して，同じ名前の新しいシンボルが作成されます。

ユーザは，パス名の中に起動番号を含めて，そのルーチンに対する特定の呼び出しを示すことができます。起動番号は，パス名の一番右側のルーチン名の後ろに非負の整数として置かなければなりません。 0 は，最も最近の起動を意味します。 1 はその前の起動，というように続きます。たとえば， PROG が COMPUTE を呼び出し，COMPUTE が自身を再帰的に呼び出しているとします。各呼び出し時に新しい変数 SUM が生成されるとき，次のコマンドは， COMPUTE の最近の呼び出し時の SUM の値を表示します。

```
DBG> EXAMINE PROG\COMPUTE 0\SUM
```

1 つ前の COMPUTE の呼び出し時に生成された変数 SUM を参照するには，上記のパス名の 0 の箇所に 1 を指定します。

起動番号を指定しないと，デバッガは，そのルーチンに対する最近の呼び出しへの参照と仮定します。省略時の起動番号は 0 です。

第 5.3.3 項で説明する SET SCOPE/CURRENT コマンドを参照してください。このコマンドを使用すれば，省略時の有効範囲検索リストの参照位置を呼び出しスタックの相対位置に再設定できます。

5.3.3 シンボル検索の有効範囲を指定する SET SCOPE の使用

省略時の設定では，デバッガはパス名接頭識別子の指定のないシンボルを検索する場合，第 5.3.1 項で説明した有効範囲検索リストを使用します。

SET SCOPE コマンドを使用すれば，シンボル検索用の新しい有効範囲を設定できます。その結果，設定された有効範囲内のシンボルを参照するときに，パス名を指定する必要がなくなります。

次の例では， SET SCOPE コマンドを使用して，シンボル検索用の新しい有効範囲としてパス名 MOD4\ROUT2 を設定しています。その後，パス名接頭識別子の指定なしで Y を参照すると，新しい有効範囲内で見える Y の宣言が使用されます。

```
DBG> EXAMINE Y
%DEBUG-E-NOUNIQUE, symbol 'Y' is not unique
DBG> SHOW SYMBOL Y
data MOD7\ROUT3\BLOCK1\Y
data MOD4\ROUT2\Y
```

```
DBG> SET SCOPE MOD4\ROUT2
DBG> EXAMINE Y
MOD4\ROUT2\Y: 12
DBG>
```

SET SCOPE コマンドによってパス名が設定されると，そのコマンド内に指定したパス名が，パス名で個別に修飾されていないすべての参照に適用されます。

SET SCOPE コマンドには，数値パス名を指定することができます。たとえば次のコマンドは，現在の有効範囲を PC 範囲から 3 コール分さかのぼったルーチン呼び出しに設定します。

```
DBG> SET SCOPE 3
```

また，有効範囲検索リストを使用すれば，デバッガがシンボルを検索する順序を指定できます。たとえば，次のコマンドによって，デバッガは最初に PC 範囲 (有効範囲 0) を検索し，次にモジュール MOD4 内のルーチン ROUT2 によって示される有効範囲を検索します。

```
DBG> SET SCOPE 0, MOD4\ROUT2
```

デバッガの省略時の有効範囲検索リストは，各有効範囲が存在すると仮定すれば次のコマンドを入力した場合と同じです。

```
DBG> SET SCOPE 0,1,2,3, . . .  ,n
```

この場合，デバッガは，シンボルを見つけるために，呼び出しスタックを順番にさかのぼって検索します。

また， SET SCOPE/CURRENT コマンドを使用して，省略時の有効範囲検索リストの参照位置を呼び出しスタック内の別のルーチンに変更することができます。たとえば，次のコマンドは，有効範囲検索リストを 2,3,4, . . . ,*n*のように設定します。

```
DBG> SET SCOPE/CURRENT 2
```

シンボル検索用の現在の有効範囲検索リストを表示するには， SHOW SCOPE コマンドを使用します。省略時の有効範囲検索リストを復元する (第 5.3.1 項を参照) には， CANCEL SCOPE コマンドを使用します。

5.4 共用可能イメージのデバッグ

省略時の設定では，ユーザ・プログラムは，弊社が提供しているいくつかの共用可能イメージ (たとえば，実行時ライブラリ・イメージ LIBRTL.EXE など) とリンクされる場合があります。この節では，ユーザ定義の共用可能イメージをデバッグするときに，前に説明した概念をどのように拡張するかについて説明します。

共用可能イメージは，直接実行することを意図したものではありません。共用可能イメージは最初に，実行可能なイメージのリンク時の入力として指定しなければなりません。リンクされた共用可能イメージは，実行可能なイメージの実行時にロードされます。共用可能イメージをデバッグするために，それをインストールする必要はありません。代わりに，論理名を割り当てることによって，プライベート・コピーをデバッグすることができます。

共用可能イメージのリンクについての詳しい説明は，『OpenVMS Linker Utility Manual』を参照してください。

5.4.1 共用可能イメージをデバッグするためのコンパイルとリンク

共用可能イメージをデバッグするためのコンパイル作業およびリンク作業の基本的な手順は，次のとおりです。

1. /DEBUG 修飾子を使用して，メイン・イメージと共用可能イメージのソース・ファイルをコンパイルする。
2. /SHAREABLE コマンド修飾子および/DEBUG コマンド修飾子を使用して共用可能イメージをリンクする。そのとき，そのイメージのユニバーサル・シンボルを宣言する。ユニバーサル・シンボルは，ある共用可能イメージ内に定義され，別のイメージから参照されるグローバル・シンボルである。
3. メイン・イメージに対して共用可能イメージをリンクする。そのとき，リンカ・オプションに/SHAREABLE ファイル修飾子を使用して，共用可能イメージを指定する。/DEBUG コマンド修飾子も指定する。
4. 共用可能イメージのローカル・コピーを指す論理名を定義する。このとき，イメージ名だけでなく装置名およびディレクトリ名も指定しなければならない。そうしないと，イメージ・アクティベータは，システムの省略時の共用可能イメージ・ライブラリ・ディレクトリである SYS$SHARE 内からその名前のイメージを検索する。
5. メイン・イメージをデバッガの制御下に置く。共用可能イメージは実行時にロードされる。

これらの手順を次の例に示します。この例では， MAIN.FOR と SUB1.FOR が，実行可能なメイン・イメージのソース・ファイルです。また， SHR1.FOR と SHR2.FOR が，デバッグされる共用可能イメージのソース・ファイルです。

第 5.1 節で説明したように，各イメージのソース・ファイルをコンパイルします。

```
$ FORTRAN/NOOPT/DEBUG MAIN,SUB1
$ FORTRAN/NOOPT/DEBUG SHR1,SHR2
```

Alpha プロセッサでは， LINK コマンドにシンボル・ベクタ・オプションを使用して，共用可能イメージを作成しユニバーサル・シンボルを指定します。次に例を示します。

```
$ LINK/SHAREABLE/DEBUG SHR1,SHR2,SYS$INPUT:/OPTIONS
SYMBOL_VECTOR=(SHR_ROUT=PROCEDURE) Ctrl/Z
```

上記のコマンド例の意味は，次のとおりです。

- /SHAREABLE コマンド修飾子は，オブジェクト・ファイル SHR1.OBJ と SHR2.OBJ から共用可能イメージ SHR1.EXE を作成する。
- SYS$INPUT: の後ろの/OPTIONS 修飾子によって，ユニバーサル・シンボル SHR_ROUT が指定可能になる。
- /DEBUG 修飾子は， SHR1.EXE のデバッグ・シンボル・テーブル (DST) とグローバル・シンボル・テーブル (GST) を作成し，それらをイメージ内に取り込む。GST には，ユニバーサル・シンボル SHR_ROUT が含まれる。

これで，共用可能イメージ SHR1.EXE が現在の省略時のディレクトリに作成されます。 SHR1.EXE は共用可能イメージなので，ユーザが明示的に実行することはありません。代わりに，実行可能なメイン・イメージとリンクします。

```
$ LINK/DEBUG MAIN,SUB1,SYS$INPUT:/OPTIONS
SHR1.EXE/SHAREABLE Ctrl/Z
$
```

上記のコマンド例の意味は，次のとおりです。

- LINK コマンドは， MAIN.OBJ と SUB1.OBJ から実行可能なイメージ MAIN.EXE を作成する。
- /DEBUG 修飾子は， MAIN.EXE の DST と GST を作成し，それらをイメージ内に取り込む。
- SHR1.EXE の後ろの/SHAREABLE 修飾子は， SHR1.EXE が MAIN.EXE の共用可能イメージとしてリンクされることを意味する。

作成されたメイン・イメージ MAIN.EXE を実行すると，リンクされたすべての共用可能イメージが，実行時にロードされます。しかし省略時の設定では，イメージ・アクティベータは，システムの省略時の共用可能イメージ・ライブラリ・ディレクトリである SYS$SHARE 内から共用可能イメージを検索します。したがって，ユーザは，論理名 SHR1 を定義して，それが現在の省略時のディレクトリ内に存在する SHR1.EXE を指すようにしなければなりません。このとき，必ず装置およびディレクトリを指定してください。

```
$ DEFINE SHR1 SYS$DISK:[]SHR1.EXE
```

これで，デバッガの RUN コマンドに MAIN を指定することによって，デバッガ起動後に MAIN と SHR1 をデバッガの制御下に置くことができます。

```
$ DEBUG/KEEP

          Debugger Banner and Version Number

DBG> RUN MAIN
```

5.4.2 共用可能イメージ内のシンボルへのアクセス

第 5.1 節，第 5.2 節，および第 5.3 節で説明したすべての概念は，単一イメージのモジュール，すなわち実行可能なメイン・イメージに適用されます。ここでは，共用可能イメージのデバッグに固有の追加情報を提供します。

第 5.4.1 項で説明したように，デバッグ用に共用可能イメージをリンクすると，リンカは各イメージに DST と GST を作成します。共用可能イメージの GST には，ユニバーサル・シンボルだけが含まれます。メモリを節約するために，デバッガはイメージが設定されたときだけ，そのイメージの RST を作成します。イメージは，動的に，または SET IMAGE コマンドの入力によって設定されます。

SHOW IMAGE コマンドは，ユーザ・プログラムにリンクされているすべての共用可能イメージを示し，設定されているイメージや現在のイメージを識別します (現在のイメージの定義については，第 5.4.2.2 項を参照)。プログラムがデバッガの制御下に置かれたときには，最初にメイン・イメージだけが設定されます。

次の各項では，デバッガが，プログラムの実行中に動的にイメージを設定する方法と，実行に関係なく任意のイメージ内のシンボルへのアクセスを可能にする方法を説明します。

インストールされた書き込み可能な共用可能イメージ内にウォッチポイントを設定する方法については，第 3.4.3.4 項を参照してください。

5.4.2.1 PC 範囲内のシンボルへのアクセス (動的モード)

省略時の設定では，動的モードが有効になっています。したがって，実行が中断したときにはいつも，停止している位置のイメージおよびモジュールが設定されます (それらのモジュールがまだ設定されていない場合)。

動的モードを使用すれば，次のようなシンボルへのアクセスが自動的に可能となります。

- 実行が停止しているイメージ内のすべての設定されているモジュールで定義されているシンボルの参照
- そのイメージの GST 内の任意のユニバーサル・シンボルの参照

SET MODULE コマンドを使用して，そのイメージ内の他のモジュールを設定すると，ユーザはそのイメージ内の任意のシンボルを参照することができます。

いったん設定されたイメージは， CANCEL IMAGE コマンドによって取り消されるまで，そのまま設定されています。設定されたイメージおよびモジュールの数が増えたため，デバッガの処理速度が低下する場合は， CANCEL IMAGE コマンドを使用してイメージを取り消します。また， SET MODE NODYNAMIC コマンドを使用して，動的モードを無効にすることもできます。

5.4.2.2 任意のイメージ内のシンボルへのアクセス

ユーザまたはデバッガが最後に設定したイメージを**現在のイメージ**と言います。現在のイメージは，シンボル検索のためのデバッグ・コンテキストです。したがって，次のコマンドを使用した場合，ユーザは現在のイメージ内に定義されているシンボルだけを参照することができます。

DEFINE/ADDRESS
DEFINE/VALUE
DEPOSIT
EVALUATE
EXAMINE
TYPE
(SET,CANCEL) BREAK
(SET,SHOW,CANCEL) MODULE
(SET,CANCEL) TRACE
(SET,CANCEL) WATCH
SHOW SYMBOL

SHOW BREAK, SHOW TRACE, SHOW WATCH の各コマンドは，すべてのイメージ内に設定されている任意のブレークポイント，トレースポイント，またはウォッチポイントを示すことに注意してください。

現在のイメージ以外のイメージ内に存在するシンボルを参照するには，SET IMAGE コマンドを使用して現在のイメージを指定したあと，SET MODULE コマンドを使用して，そのシンボルが定義されているモジュールを設定します。SET IMAGE コマンドは，モジュールを設定しません。次のサンプル・プログラムは，これらの概念を示したものです。

サンプル・プログラムは，メイン・イメージ PROG1 と共用可能イメージ SHR1 からなります。現在，プログラムがデバッガの制御下に置かれ，実行がイメージ PROG1 内のメイン・プログラム単位で停止していると想定します。ここで，ルーチン ROUT2 にブレークポイントを設定したいとします。ROUT2 は，イメージ SHR1 内のあるモジュールで定義されています。

ROUT2 にブレークポイントを設定しようとすると，デバッガは現在のイメージ PROG1 内から ROUT2 を検索します。

```
DBG> SET BREAK ROUT2
%DEBUG-E-NOSYMBOL, symbol 'ROUT2' is not in symbol table
DBG>
```

SHOW IMAGE コマンドは，イメージ SHR1 の設定が必要であることを示しています。

```
DBG> SHOW IMAGE
 image name           set    base address    end address

*PROG1                yes    00000200        000009FF
 SHR1                 no     00001000        00001FFF

 total images: 2           bytes allocated: 32856
DBG> SET IMAGE SHR1

DBG> SHOW IMAGE
 image name           set    base address    end address

 PROG1                yes    00000200        000009FF
*SHR1                 yes    00001000        00001FFF

 total images: 2           bytes allocated: 41948
DBG>
```

これで，SHR1 が設定され，現在のイメージになりました。しかし，SET IMAGE コマンドはモジュールを設定しないので，ユーザはブレークポイントを設定する前に，ROUT2 が定義されているモジュールを設定しなければなりません。

```
DBG> SET BREAK ROUT2
%DEBUG-E-NOSYMBOL, symbol 'ROUT2' is not in symbol table
DBG> SET MODULE/ALL
DBG> SET BREAK ROUT2
DBG> GO
break at routine ROUT2
10:     SUBROUTINE ROUT2(A,B)
DBG>
```

イメージ SHR1 とそのすべてのモジュールが設定され，ROUT2 でブレークポイントに到達しています。あとは，通常の方法でデバッグすることができます。たとえば，ルーチン内の命令のステップ実行や変数の検査などです。

いったん設定されたイメージおよびそのイメージ内のモジュールは，新しい現在のイメージが設定されても，そのまま設定されています。しかし，任意の一時点に複数のシンボルにアクセスできるのは，それらのシンボルが現在のイメージ内に存在するときだけです。

5.4.2.3 実行時ライブラリおよびシステム・イメージ内のユニバーサル・シンボルへのアクセス

次の段落では，実行時ライブラリ，またはシンボル・テーブル情報が生成されていないその他の共用可能イメージ内のユニバーサル・シンボル (ルーチン名など) にアクセスする方法について説明します。この方法を使用すれば，ユーザはたとえば CALL コマンドを使用して，第 13.7 節で説明するように実行時ライブラリまたはシステム・サービス・ルーチンを実行できます。

次のコマンド構文を使用して SET MODULE コマンドを入力します。

```
SET MODULE SHARE$image-name
```

次に例を示します。

```
DBG> SET MODULE SHARE$LIBRTL
```

デバッガは，ユーザ・プログラム内の各共用可能イメージに仮モジュールを作成します。これらの共用可能イメージ・モジュールの名前には，接頭辞"SHARE$"が付いています。 SHOW MODULE/SHARE コマンドは，現在のイメージ内のモジュールだけでなく，これらの共用可能イメージ・モジュールも示します。

SET MODULE コマンドによって，いったん共用可能イメージ・モジュールが設定されると，ユーザは，そのイメージ内のすべてのユニバーサル・シンボルにアクセスできます。次のコマンドは， LIBRTL 内のすべてのユニバーサル・シンボルをリストします。

```
DBG> SHOW SYMBOL * IN SHARE$LIBRTL
   .
   .
   .
routine SHARE$LIBRTL\STR$APPEND
routine SHARE$LIBRTL\STR$DIVIDE
routine SHARE$LIBRTL\STR$ROUND
   .
   .
   .
routine SHARE$LIBRTL\LIB$WAIT
routine SHARE$LIBRTL\LIB$GETDVI
   .
   .
   .
```

ユーザは，これらのユニバーサル・シンボルを，たとえば CALL コマンドや SET BREAK コマンドなどに指定できます。

SET MODULE コマンドを使用して共用可能イメージ・モジュールを設定すると，そのイメージのユニバーサル・シンボルが実行時シンボル・テーブルにロードされます。その結果，これらのシンボルは現在のイメージから参照できるようになります。しかし，そのイメージ内の他のシンボル (ローカル・シンボルやグローバル・シンボル) は，現在のイメージからは参照できません。すなわち，ユーザのデバッグ・コンテキストは，現在のイメージに設定されたままです。

5.4.3 常駐イメージのデバッグ (Alpha のみ)

常駐イメージは，効率を高めることができるように特定の方法で作成され，インストールされた共用可能モジュールです。このようなイメージを作成するには，シンボル・テーブルを除いてイメージをリンクし，システム空間でイメージを実行する必要があります。このようにして作成したイメージは，デバッグが困難になります。次の手順では，もっと簡単にデバッグできる常駐イメージを作成します。

1. 共用可能イメージをコンパイルする。次の例を参照。

```
$ CC/DEBUG/NOOPTIMIZE RESIDENTMODULE.C
```

2. /DSF 修飾子を使用して共用可能イメージをリンクする。次の例を参照。

```
$ LINK/NOTRACEBACK/SHAREABLE/SECTION_BINDING/DSF RESIDENTMODULE
```

   イメージのリンクについては，『OpenVMS Linker Utility Manual』を参照。

3. インストール済み常駐イメージを作成する。 Install ユーティリティの使い方については，『OpenVMS システム管理ユーティリティ・リファレンス・マニュアル (上巻)』を参照。常駐イメージの詳細については，『OpenVMS システム管理者マニュアル (下巻)』を参照。

4. 常駐イメージを呼び出すプログラムをコンパイルする。次の例を参照。

```
$ CC/DEBUG/NOOPTIMIZE TESTPROGRAM
```

5. 常駐イメージを呼び出す実行可能イメージを作成する。次の例を参照。

```
$ LINK/DSF TESTPROGRAM
```

6. 常駐イメージのプライベート・コピーを作成する。次の例を参照。

```
$ COPY SYS$LIBRARY:RESIDENTMODULE.EXE []RESIDENTMODULE.EXE
```

7. 常駐イメージのプライベート・コピーを指す論理名を定義する。次の例を参照。

```
$ DEFINE RESIDENTMODULE []RESIDENTMODULE
```

8. テスト・プログラム用の.DSF ファイルと常駐モジュール用の.DSF ファイルの両方が同じディレクトリに格納されていることを確認する。

9. .DSF ファイルを格納したディレクトリを指すように， DBG$IMAGE_DSF_PATH を定義する。

10. デバッガを起動する。次の例を参照。

```
$ DEBUG/KEEP TESTPROGRAM
```

これで実行可能イメージと常駐イメージに対して，すべてのデバッグ・オプションを使用できるようになります。

6
ソース・コードの表示の制御

ソース・コードとは，ソース・ファイル内に現れるプログラミング言語の文を指します。ソース・コードの各行はソース行と呼ばれます。

本章には次の内容が含まれています。

- ソース・ファイルとソース行に関する情報を取得する
- コンパイル後に別のディレクトリに移動したソース・ファイルの記憶位置の指定
- 行番号，コード・アドレス式，または検索文字列を指定することによるソース行の表示
- ブレークポイント，トレースポイント，およびウォッチポイントでのソース・コードの表示と， STEP コマンドの実行後のソース・コードの表示を制御する方法
- 一定の環境でのソース行の表示を改善するための SET MARGINS コマンドの使用方法

本章で説明する方法は，行 (非画面) モードだけでなく画面モードにも適用できます。行モードと画面モードの動作内容の相違点は，本章と各コマンドの説明の中で示します。画面モードについての詳細な説明は，第 7 章を参照してください。

プログラムがコンパイラによって最適化されている場合は，デバッグ時に実行されるコードがソース・コードに一致しないこともあります。詳しい説明は，第 14.1 節を参照してください。

6.1 デバッガがソース・コード情報を取得する方法

コンパイラは，オブジェクト・モジュールを生成するためにソース・ファイルを処理する場合，各ソース行に順に行番号を割り当てます。ほとんどの言語の場合，各コンパイル単位 (モジュール) は行 1 から始まります。 Ada などでは各ソース・ファイルが行 1 から始まり， 1 つのソース・ファイルが複数のコンパイル単位を表す場合もあります。

行番号は/LIST コンパイル・コマンド修飾子を使用して取得したソース・リスト内に表示されます。また，デバッガがソース・コードを表示する際に，行モードと画面モードのどちらの場合でも常に行番号が表示されます。さらに，いくつかのデバッガ・コマンド (たとえば， TYPE および SET BREAK) で行番号を指定することができます。

コンパイル・コマンドとLINKコマンドの両方に/DEBUGコマンドを指定した場合にだけ，デバッグ時にソース行は表示されます。そのような条件下では，コンパイラによって作成されてデバッグ・シンボル・テーブル(DST)に渡されたシンボル情報には，ソース行コリレーション・レコードが含まれます。ある特定のモジュールでは，ソース行コリレーション・レコードには，そのモジュールを支える各ソース・ファイルの完全なファイル指定が入っています。また，ソース行コリレーション・レコードはソース・レコード(シンボルや型など)をモジュール内のソース・ファイルと行番号に対応づけます。

6.2 ソース・ファイルの記憶位置の指定

デバッグ・シンボル・テーブル(DST)には，各ソース・ファイルのコンパイル時における完全なファイル指定が入っています。したがって，省略時の設定では，デバッガはソース・ファイルがコンパイル時と同じディレクトリの中に入っているものと想定します。ソース・ファイルをコンパイル後に別のディレクトリに移動した場合，デバッガはそれを見つけることができず，そのファイルからソース・コードを表示しようとすると次のような警告を表示します。

```
%DEBUG-W-UNAOPNSRC, unable to open source file DISK:[JONES.WORK]PRG.FOR;2
```

このような場合，デバッガに新しいディレクトリを指示するため， SET SOURCEコマンドを使用します。このコマンドは，そのプログラム用のすべてのソース・ファイルに対して適用したり，特定のモジュール用のソース・ファイルだけに適用したりできます。

たとえば，次のコマンド行を入力すると，デバッガはWORK$:[JONES.PROG3]:の中からすべてのソース・ファイルを探します。

```
DBG> SET SOURCE WORK$:[JONES.PROG3]
```

ディレクトリ検索リストをSET SOURCEコマンドで指定できます。たとえば，次のコマンド行を入力すると，デバッガはまず現在の省略時ディレクトリ([])からファイルを探し，次にWORK$:[JONES.PROG3]:を探します。

```
DBG> SET SOURCE [], WORK$:[JONES.PROG3]
```

ある特定のモジュール用のソース・ファイルだけにSET SOURCEコマンドを適用したい場合は，/MODULE=*module-name*修飾子を使用してそのモジュールを指定します。たとえば，次のコマンド行は，モジュールSCREEN_IO用のソース・ファイルがディレクトリDISK2:[SMITH.SHARE]に入っていることを指定します。他のモジュール用のソース・ファイルの検索は，このコマンドによって影響を受けません。

```
DBG> SET SOURCE/MODULE=SCREEN_IO DISK2:[SMITH.SHARE]
```

つまり， SET SOURCE/MODULE コマンドは特定のモジュール用のソース・ファイルの記憶位置を指定し， SET SOURCE コマンドは SET SOURCE/MODULE コマンド内で明示的に指定されなかったモジュール用のソース・ファイルの記憶位置を指定します。

SET SOURCE コマンドを入力する場合，/LATEST か/EXACT の修飾子が常に有効であることを確認してください。/LATEST は，最新バージョン (ディレクトリ内にある最大番号のバージョン) のソース・ファイルを検索するようデバッガに指示します。/EXACT 修飾子は，最後にコンパイルしたバージョン (コンパイル時に作成されたデバッガ・シンボル・テーブルに記録されているバージョン) を検索するようデバッガに指示します。たとえば， SET SOURCE/LATEST コマンドは SORT.FOR;3 を検索し， SET SOURCE/EXACT は SORT.FOR;1 を検索するという具合です。

SHOW SOURCE コマンドは，現在有効なすべてのソース・ディレクトリ検索リストを表示するために使用します。このコマンドは 1 つまたは複数の SET SOURCE /MODULE コマンドで前に設定したとおりの特定のモジュール用の検索リストを表示し， SET SOURCE コマンドで前に設定したとおりの他のすべてのモジュール用の検索リストを表示します。次に例を示します。

```
DBG> SET SOURCE [PROJA],[PROJB],USER$:[PETER.PROJC]
DBG> SET SOURCE/MODULE=COBOLTEST [], DISK$2:[PROJD]
DBG> SHOW SOURCE
source directory search list for COBOLTEST:
        []
        DISK$2:[PROJD]
source directory search list for all other modules:
        [PROJA]
        [PROJB]
        USER$:[PETER.PROJC]
DBG>
```

SET SOURCE コマンドも SET SOURCE/MODULE コマンドも入力しなかった場合， SHOW SOURCE コマンドは現在有効な検索リストがないことを示します。

前の SET SOURCE コマンドの効力を取り消すには， CANCEL SOURCE コマンドを使用します。前の SET SOURCE/MODULE コマンドの効力を取り消すには，CANCEL SOURCE/MODULE コマンドを同じモジュール名を指定して使用します。

ソース・ディレクトリ検索リストを取り消した場合，デバッガは再び，指定されたモジュールに対応するソース・ファイルがコンパイル時と同じディレクトリに入っているものと想定します。

コンパイル後に別のディレクトリに移されたソース・ファイルをデバッガが検索する方法についての詳しい説明は， SET SOURCE コマンドの説明を参照してください。

6.3 行番号の指定によるソース・コードの表示

TYPE コマンドを使用すると，コンパイラ割り当て行番号を指定することによってソース行を表示できます。その場合，それぞれの行番号はソース・コードの 1 行を指定します。

たとえば，次のコマンドはデバッグ中のモジュールの行 160 と行 22 ～ 24 を表示します。

```
DBG> TYPE 160, 22:24
module COBOLTEST
   160: START-IT-PARA.
module COBOLTEST
    22: 02      SC2V2   PIC S99V99      COMP VALUE  22.33.
    23: 02      SC2V2N  PIC S99V99      COMP VALUE -22.33.
    24: 02      CPP2    PIC PP99        COMP VALUE  0.0012.
DBG>
```

あるモジュールのすべてのソース行を表示するには， 1 から始めてそのモジュール内の最大の行番号またはそれ以上の数で終わる行番号の範囲を指定します。

ソース行の表示後は，行番号なしの TYPE コマンドを入力すれば，つまり TYPE コマンドを入力してから Return キーを押せば，そのモジュール内の次の行を表示できます。次に例を示します。

```
DBG> TYPE 160
module COBOLTEST
   160: START-IT-PARA.
DBG> TYPE
module COBOLTEST
   161:         MOVE SC1 TO ES0.
DBG>
```

その後，繰り返し TYPE コマンドを入力することによって，その次の行とそれ以降の行を表示できます。このようにして，一度に 1 行ずつコードを参照することができます。

プログラム内の任意のモジュールのソース行を表示するには，行番号といっしょにモジュール名を指定します。パス名の表記法は標準のものを使用します。つまり，最初にモジュール名を指定し，次にバックスラッシュ(\)，最後に行番号または行番号の範囲を指定しますが，それらの間にスペースは入れません。たとえば，次のコマンドは TEST というモジュールの行 16 を表示します。

```
DBG> TYPE TEST\16
```

TYPE コマンドでモジュール名を指定する場合，そのモジュールは設定されていなければなりません。特定のモジュールが設定されているかどうかを判別するには，SHOW MODULE コマンドを使用します。その後，必要であれば SET MODULE コマンドを使用してください (第 5.2 節を参照)。

TYPE コマンドでモジュール名を指定しなかった場合，デバッガは省略時の設定では現在実行が一時停止されているモジュール，つまり， PC 範囲に対応したモジュールのソース行を表示します。 SET SCOPE コマンドで別の有効範囲を指定してある場合は，デバッガは指定された有効範囲に対応するモジュールのソース行を表示します。

画面モードでは， TYPE コマンドの出力は現在のソースの表示を更新します (第 7.2.6 項を参照)。

プログラム内のさまざまな記憶位置のソース行を表示したあと， KP5 を押すと，現在実行が一時停止されている行を再表示することができます。

6.4 コード・アドレス式の指定によるソース・コードの表示

EXAMINE/SOURCE コマンドを使用すると，コード・アドレス式に対応したソース行を表示できます。コード・アドレス式は機械語コード命令のアドレスを表すものなので，次のいずれかでなければなりません。

- 1 つまたは複数の命令に対応した行番号
- ラベル
- ルーチン名
- 1 つの命令のメモリ・アドレス

EXAMINE/SOURCE コマンドに変数名を指定することはできません。変数名は，命令ではなくデータに対応しているからです。

EXAMINE/SOURCE コマンドを使用した場合，デバッガはメモリ・アドレスを取得するためにアドレス式を評価し，どのコンパイラ割り当て行番号がそのアドレスに対応するかを判別してから，その行番号によって指定されるソース行を表示します。

たとえば，次のコマンド行はルーチン SWAP のアドレス (宣言) に対応したソース行を表示します。

```
DBG> EXAMINE/SOURCE SWAP
module MAIN
    47:  procedure SWAP(X,Y: in out INTEGER) is
DBG>
```

命令に対応しない行番号を指定した場合，デバッガは診断メッセージを発行します。次に例を示します。

```
DBG> EXAMINE/SOURCE %LINE 6
%DEBUG-I-LINEINFO, no line 6, previous line is 5, next line is 8
%DEBUG-E-NOSYMBOL, symbol '%LINE 6' is not in the symbol table
DBG>
```

EXAMINE/SOURCE コマンドをシンボリック・アドレス式 (行番号，ラベル，またはルーチン) といっしょに使用する際，その要素を定義しているモジュールがまだ設定されていないときには，それを設定しなければならない場合もあります。特定のモジュールが設定されているかどうかを判別するには， SHOW MODULE コマンドを使用します。その後，必要であれば SET MODULE コマンドを使用します (第 5.2 節を参照)。

EXAMINE/SOURCE .%PC コマンドは，現在の PC 値 (実行されようとしている行) に対応するソース行を表示します。次に例を示します。

```
DBG> EXAMINE/SOURCE .%PC
module COBOLTEST
   162:         DISPLAY ES0.
DBG>
```

内容演算子(.)の使用に注意してください。この演算子は，このピリオドの後ろにある要素の内容を指定します。内容演算子を使用しなかった場合，デバッガは PC 内に現在格納されているアドレスではなく， PC のソース行を見つけようとします。

```
DBG> EXAMINE/SOURCE %PC
%DEBUG-W-NOSRCLIN, no source line for address 7FFF005C
DBG>
```

次の例は，数値パス名 (1\) を使用して呼び出しスタックの 1 レベル下の PC 値にある実行が一時停止されているルーチンの呼び出しのソース行を表示する例です。

```
DBG> EXAMINE/SOURCE .1\%PC
```

画面モードでは， EXAMINE/SOURCE コマンドの出力は現在のソースの表示を更新します (第 7.2.6 項を参照)。

デバッガは，次のコンテキストで EXAMINE/SOURCE コマンドを使用して現在の PC 値のソース・コードを表示します。

キーパッド・キー 5 (KP5) が次のような一連のデバッガ・コマンドにバインドされています。

```
EXAMINE/SOURCE .%SOURCE_SCOPE\%PC; EXAMINE/INST .%INST_SCOPE\%PC
```

この一連のコマンドは，現在の有効範囲内で現在実行が一時停止されている位置のソース行と命令を表示します。 KP5 を押せばデバッグ・コンテキストを素早く判別することができます。

定義済みソース・ディスプレイ「SRC」は自動的に更新される表示です。これは，デバッガが実行に割り込みをかけ，コマンドを求めるプロンプトを表示するたびに，次の組み込みコマンドを実行します (第 7.4.1 項を参照)。

```
EXAMINE/SOURCE .%SOURCE_SCOPE\%PC
```

6.5 文字列の検索によるソース・コードの表示

SEARCH コマンドを使用すると，指定した文字列を含んでいるソース行を表示できます。

SEARCH コマンドの構文は次のとおりです。

```
SEARCH[/qualifier[, ... ]] [range] [string]
```

範囲のパラメータには，モジュール名，行番号の範囲，またはそれらの組み合わせが指定できます。モジュール名を指定しなかった場合，デバッガは TYPE コマンドの場合と同様に現在の有効範囲を使用してソース行を見つけます (第 6.3 節を参照)。

省略時の設定では， SEARCH コマンドは指定された範囲内でその文字列が最初に (次に) 現れるソース行を表示します (SEARCH/NEXT)。 SEARCH/ALL コマンドは，指定された範囲内でその文字列が現れるすべてのソース行を表示します。たとえば，次のコマンド行はモジュール SCREEN_IO 内で pro という文字列が最初に現れるソース行を表示します。

```
DBG> SEARCH SCREEN_IO pro
```

以下の例では， 1 つの COBOL モジュールに入っているソース行を使用して，現在の有効範囲内で SEARCH コマンドのさまざまな側面を示しています。

次のコマンド行は，文字列 D を含んでいるすべてのソース行を行 40 ～ 50 から抽出して表示します。

```
DBG> SEARCH/ALL 40:50 D
module COBOLTEST
    40: 02      D2N     COMP-2 VALUE -234560000000.
    41: 02      D       COMP-2 VALUE  222222.33.
    42: 02      DN      COMP-2 VALUE -222222.333333.
    47: 02      DR0     COMP-2 VALUE  0.1.
    48: 02      DR5     COMP-2 VALUE  0.000001.
    49: 02      DR10    COMP-2 VALUE  0.00000000001.
    50: 02      DR15    COMP-2 VALUE  0.0000000000000001.
DBG>
```

特定のモジュール内で文字列の発生箇所を見つけたあと，パラメータを付けずに SEARCH コマンドを入力すれば，同じモジュール内で同じ文字列が次に現れるソース行を表示することができます。これは，パラメータを付けずに TYPE コマンドを使用して次のソース行を表示するのと似ています。次に例を示します。

```
DBG> SEARCH 42:50 D
module COBOLTEST
    42: 02      DN      COMP-2 VALUE -222222.333333.
DBG> SEARCH
module COBOLTEST
    47: 02      DR0     COMP-2 VALUE  0.1.
DBG>
```

省略時の設定では，デバッガは指定されたとおりの文字列を検索し，その文字列の前後のコンテキストは解釈しません。これが SEARCH/STRING の動作内容です。ある文字列がプログラム内で識別子 (たとえば，変数名) として使用されている箇所を検索し，識別子以外の箇所を除外したい場合は，/IDENTIFIER 修飾子を使用します。SEARCH/IDENTIFIER コマンドは，その文字列の両端が現在の言語で識別子の一部とはならない文字によって区切られている場合にだけその文字列を表示します。

SEARCH コマンドの省略時の修飾子は/NEXT と/STRING です。別の省略時の修飾子を設定したい場合は SET SEARCH コマンドを使用します。たとえば，次のコマンドの実行後は， SEARCH コマンドは SEARCH/IDENTIFIER のように動作します。

```
DBG> SET SEARCH IDENTIFIER
```

現在 SEARCH コマンドに有効な省略時の修飾子を表示するには， SHOW SEARCH コマンドを使用します。次に例を示します。

```
DBG> SHOW SEARCH
search settings: search for next occurrence, as an identifier
DBG>
```

6.6 ステップ実行後，およびイベントポイントでのソースの表示の制御

省略時の設定では，対応するソース行をデバッガが表示するのは，ブレークポイント，トレースポイント，ウォッチポイントの検出後，または 1 つの STEP コマンドが完了したときです。

STEP コマンドを入力した場合，デバッガはステップ実行後に実行が一時停止した位置のソース行を表示します。次に例を示します。

```
DBG> STEP
stepped to MAIN\%LINE 16
    16:         RANGE := 500;
DBG>
```

ブレークポイントまたはトレースポイントが検出された場合，デバッガはそのブレークポイントまたはトレースポイントの位置のソース行を表示します。次に例を示します。

```
DBG> SET BREAK SWAP
DBG> GO
   .
   .
   .
break at MAIN\SWAP
    47: procedure SWAP(X,Y: in out INTEGER) is
DBG>
```

ウォッチポイントが検出された場合，デバッガはそのウォッチポイントが検出される原因となった命令に対応するソース行を表示します。

SET STEP [NO]SOURCE コマンドを使用すると， 1 ステップの実行後のソース・コードの表示，またはブレークポイント，トレースポイント，ウォッチポイントでのソース・コードの表示を制御することができます。 SET STEP SOURCE (省略時の設定) はソースの表示を有効にします。 SET STEP NOSOURCE はソースの表示を無効にします。次に例を示します。

```
DBG> SET STEP NOSOURCE
DBG> STEP
stepped to MAIN\%LINE 16
DBG> SET BREAK SWAP
DBG> GO
   .
   .
   .
break at MAIN\SWAP
DBG>
```

SET STEP SOURCE コマンドまたは SET STEP NOSOURCE コマンドの効力を選択的に上書きするには， STEP, SET BREAK, SET TRACE，および SET WATCH の各コマンドに/SOURCE および/NOSOURCE の修飾子を使用します。

STEP/SOURCE コマンドは SET STEP NOSOURCE コマンドの効力を上書きしますが，それはその STEP コマンドが実行される間だけです。同様に， STEP /NOSOURCE も SET STEP SOURCE の効力をその STEP コマンドが実行される間だけ上書きします。次に例を示します。

```
DBG> SET STEP NOSOURCE
DBG> STEP/SOURCE
stepped to MAIN\%LINE 16
    16:        RANGE := 500;
DBG>
```

SET BREAK/SOURCE コマンドは SET STEP NOSOURCE コマンドの効力を上書きしますが，それはその SET BREAK コマンドによって設定されたブレークポイントに対してだけです。同様に， SET BREAK/NOSOURCE も SET STEP SOURCE の効力をその SET BREAK コマンドによって設定されたブレークポイントに対してだけ上書きします。この規則は SET TRACE と SET WATCH にも適用されます。次に例を示します。

```
DBG> SET STEP SOURCE
DBG> SET BREAK/NOSOURCE SWAP
DBG> GO
   .
   .
   .
break at MAIN\SWAP
DBG>
```

6.7 ソースの表示用のマージンの設定

SET MARGINS コマンドを使用すると，ソース行の表示を開始し終了する左端と右端のソース行文字位置 (左マージンと右マージン) が指定できます。これは，たとえばコードが深くインデントされていたり，長い行が右マージンで折り返したりする場合に，ソース・コードの表示を制御するのに役立ちます。このような場合には，インデントされたスペースをソースの表示で削除するために左マージンを設定したり，行を切り捨てて折り返しを防ぐために右マージンを減らしたりできます。

たとえば，次のコマンド行は左マージンを桁 25，右マージンを桁 35 に設定します。

```
DBG> SET MARGINS 20:35
```

このあとでソース行を表示するコマンド (たとえば， TYPE, SEARCH, STEP) を入力すると，ソース・コードの桁 20 と桁 35 の間の部分だけが表示されます。ソース行表示の現在のマージン設定を示すには， SHOW MARGINS コマンドを使用します。

SET MARGINS コマンドはソース行の表示だけにしか影響を及ぼさないので注意してください。このコマンドは，その他のデバッガ出力 (たとえば EXAMINE コマンドによる出力など) の表示には影響しません。

ほとんどの場合， SET MARGINS コマンドは行 (非画面) モードで役立ちます。画面モードでは， SET MARGINS コマンドは定義済みディスプレイ「SRC」などのソースの表示におけるソース行の表示には効力を持ちません。

7
画面モード

画面モードは， OpenVMS デバッガのコマンド行インタフェースの拡張機能であり，このモードでは， DECwindows Motif for OpenVMS ユーザ・インタフェースの場合と同じ方法で (第 3 部を参照)，デバッグ・セッションに関する個別のデータ・グループを同時に表示できます。たとえば，画面の一部分にソース・コードを表示し，別の部分にレジスタの内容を表示し，さらに別の部分にデバッガからの出力を表示できます。

画面モードを起動するには，キーパッドの PF3 を押します (または SET MODE SCREEN コマンドを入力します)。コマンド行インタフェースによるデバッグに戻るには， PF1 PF3 を押します (または SET MODE NOSCREEN コマンドを入力します)。

注意

デバッガに対する DECWindows Motif **インタフェースの内部から画面モードを開始することはできません。**

画面モード出力は， VT52 以上の VT シリーズ端末，および VWS を使用しているワークステーションの場合に最適です。特に，ワークステーションの画面が大きいほど，多数の画面をさまざまな目的に使用するのに適しています。

本章には次の内容が含まれています。

- 本章全体で使用される画面モードの概念と用語
- 各種のディスプレイの使用方法
- ディスプレイ属性の割り当てによってデバッガ出力を各種のディスプレイへ出力する方法
- 画面モードになった時点で自動的に使用できるようになる定義済みディスプレイの SRC， OUT， PROMPT， INST， REG， IREG，および FREG (Alpha のみ) の使用方法
- ディスプレイのスクロール，非表示，削除，移動，およびサイズ変更
- 新しいディスプレイの作成
- ディスプレイ・ウィンドウの指定
- ディスプレイ構成の作成
- 画面ディスプレイの現在の状態の保存

- デバッグ・セッション中に端末画面の高さと幅を変更する方法とディスプレイ・ウィンドウでの効果
- 画面に関連したデバッガ組み込みシンボルの使用方法
- 定義済みウィンドウの使用方法
- 画面モードの各国固有の機能を使用可能にする

画面モード・コマンドの多くはキーパッド・キーにバインドされています。キー定義については，付録 A を参照してください。

注意

本章では，1 つまたは複数のプロセスで実行されるプログラムに共通した情報を提供します。その他のマルチプロセス・プログラム固有の情報については，第 15 章を参照してください。

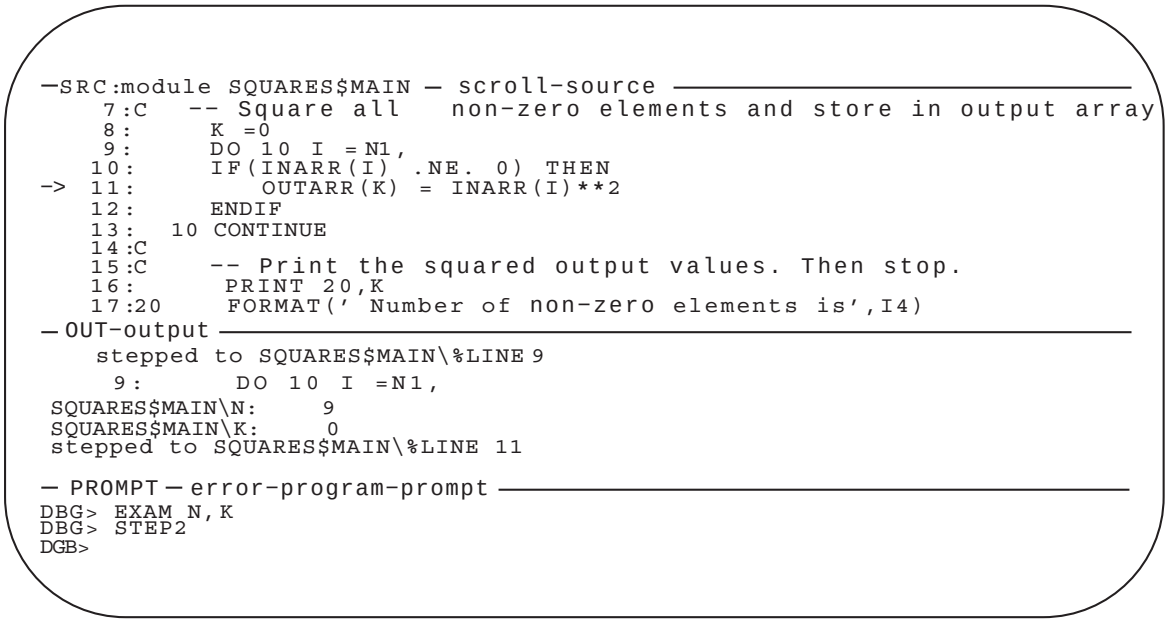
7.1 概念と用語

ディスプレイとはテキスト行のまとまりのことです。テキストには，ソース・ファイルに入っている行，アセンブリ言語の命令，レジスタに入っている値，ユーザからデバッガへの入力，デバッガ出力，またはプログラム入出力 (I/O) などがあります。

ディスプレイは**ディスプレイ・ウィンドウ**を通して参照します。ウィンドウは画面の任意の長方形の領域を占有できます。ディスプレイ・ウィンドウはディスプレイそのものより小さいのが普通なので，ディスプレイ・テキストを越えてウィンドウを上下左右にスクロールさせ，ディスプレイの任意の部分を参照することができます。

図 7–1 は， 3 つのディスプレイ・ウィンドウのある画面モードの例です。各ディスプレイの名前 (SRC, OUT，および PROMPT) は，それぞれのウィンドウの左上隅にあります。これらはディスプレイ自体のタグとして機能するだけでなく，あとでコマンドで参照するための名前としても機能します。

図 7–1 省略時の画面モード・ディスプレイ構成



ZK–6503–GE

図 7–1 は，画面モードを最初に起動したときに設定される省略時のディスプレイ構成です。 SRC, OUT，および PROMPT は，画面モードに入ったときに省略時の設定としてデバッガが提供する 3 つの**定義済みディスプレイ** (第 7.4 節を参照) です。追加のディスプレイを作成するのと同様にこれらのディスプレイの構成を変更することができます。

SRC， OUT，および PROMPT の各ディスプレイには次の基本的な特性があります。

- SRC は画面の上半分を使用するソース・コードのディスプレイである (図 7–1 の例では， Fortran コードが表示されている)。表示されるソース・モジュールの名前， SQUARES$MAIN はディスプレイ名の右側に表示される。
- SRC のすぐ下のウィンドウにあるディスプレイ OUT は，デバッガ・コマンドの出力を示している。
- 画面の一番下にあるディスプレイ PROMPT は，デバッガのプロンプトと入力を示している。

概念上では，ディスプレイは**ペーストボード**上の場合と同じように画面上に配置されます。あるコマンドによって参照された最新のディスプレイは，省略時の設定ではペーストボードの最上部に置かれます。したがって，ウィンドウの記憶位置によって，最後に参照したディスプレイが他のディスプレイに重なるか他のディスプレイを隠すかが決まります (ペーストボード上の場合と同じ)。

デバッガは，ディスプレイをペーストする順序である**ディスプレイ・リスト**を保持します。いくつかのキーパッド・キー定義は，現在ペーストボード上にあるディスプレイの間を循環するために，このディスプレイ・リストを使用します。

ディスプレイはすべて何らかの**ディスプレイ対象**に所属します (第 7.2 節を参照)。ディスプレイがどのような種類の情報 (たとえば，ソース・コード，アセンブリ言語命令，各種のデバッガ出力など) を取り込んで表示できるかは，このディスプレイ対象によって決まります。またディスプレイ対象によって，ディスプレイの内容が生成される方法も決まります。

ディスプレイの内容は 2 つの方法で生成されます。

- 自動的に更新されるディスプレイがいくつかあります。それらのディスプレイの定義には，デバッガがプログラムから制御を取得したときに常に実行されるコマンド・リストが含まれています。このコマンド・リストの出力が，自動更新ディスプレイの内容になります。ディスプレイ SRC はこのカテゴリに属します。このディスプレイは，ウィンドウの中央に置かれた矢印が，現在実行が一時停止している位置のソース行を示すように自動的に更新されます。
- もう 1 つのディスプレイ，たとえば，ディスプレイ OUT などでは，ユーザが会話形式で入力したコマンドの内容がディスプレイ内容になります。この一般的なカテゴリに属すディスプレイを作成する場合，ユーザは前もってそのディスプレイを 1 つまたは複数の種類の出力用のターゲット・ディスプレイとして選択 (SELECT コマンドを使用) しておかなければ，そのディスプレイに書き込みを行うことはできません。このことを，ディスプレイへの 1 つまたは複数の**ディスプレイ属性**の割り当てといいます (第 7.3 節を参照)。

ディスプレイへ割り当てた属性の名前は，ディスプレイ名の右側に小文字で表示されます。たとえば図 7–1 では， SRC は source 属性と scroll 属性を持ち (SRC は**現在のソースのディスプレイ**であり**現在のスクロール・ディスプレイ**である)， OUT は output 属性を持っています。これは**現在の出力ディスプレイ**です。 SRC はその独自の組み込みコマンドによって自動的に更新されますが， source 属性を持っているので，ある種の会話型コマンド (たとえば， EXAMINE/SOURCE など) の出力も受け取ることに注意してください。

この節で紹介した概念については，この章の残りの部分でさらに詳しく説明します。

7.2 ディスプレイ対象

ディスプレイはすべてディスプレイ対象を持っています。このディスプレイ対象は，そのディスプレイに含まれる情報の種類とその情報が生成される方法を決めます。またディスプレイに関連するメモリ・バッファがページングされるかどうかを決めます。

通常，ディスプレイ対象は新しいディスプレイを作成するために DISPLAY コマンドを使用するときに指定します。ディスプレイ対象を指定しなかった場合は**出力ディスプレイ**が作成されます。 DISPLAY コマンドは，次のキーワードと一緒に使用して，既存のディスプレイのディスプレイ対象を変更するために使用することもできます。

DO (*command[,...]*)
INSTRUCTION
INSTRUCTION (*command*)
OUTPUT
REGISTER
SOURCE
SOURCE (*command*)

レジスタ・ディスプレイの内容はデバッガによって自動的に生成および更新されます。その他のディスプレイの対象の内容はコマンドによって生成され，それらのディスプレイ対象は 2 つの一般的なグループに分かれます。

次のディスプレイ対象のいずれかに属するディスプレイでは，ユーザがそのディスプレイを定義したときに与えたコマンドまたはコマンド・リストに従って内容が自動的に更新されます。

DO (*command[,...]*)
INSTRUCTION (*command*)
REGISTER
SOURCE (*command*)

指定したコマンド・リストは，そのディスプレイが除去済みとしてマークされていないかぎり，デバッガがユーザのプログラムから制御を受け取るたびに実行されます。それらのコマンドの出力がディスプレイの新しい内容になります。ディスプレイが除去済みとしてマークされている場合，デバッガはユーザがそのディスプレイを可視にする (そのディスプレイを非除去済みとしてマークする) まで，そのコマンドを実行しません。

次のディスプレイ対象のいずれかに属するディスプレイでは，ユーザが会話形式で入力したコマンドから内容が作成されます。

INSTRUCTION
OUTPUT
SOURCE

デバッガの出力をこのグループの特定のディスプレイへ出力するには，まず SELECT コマンドでそのディスプレイを選択しなければなりません。その方法については次の各項と，第 7.3 節で説明します。特定の出力のディスプレイを選択したあとは，ユーザのコマンドからの出力がそのディスプレイの内容になります。

7.2.1 DO (コマンド[; . . .]) ディスプレイ対象

DO ディスプレイは自動的に更新されるディスプレイです。コマンド・リスト内のコマンドは，デバッガがユーザのプログラムから制御を受け取るたびに，リストされた順に実行されます。それらのコマンドの出力がディスプレイの内容になり，前の内容は消去されます。

たとえば，次のコマンドは DO ディスプレイの CALLS をウィンドウ Q3 に作成します。(ウィンドウ Q3 は，ウィンドウの画面寸法を示します。画面寸法と，定義済みのウィンドウについては，第 7.12 節を参照してください。) デバッガがプログラムから制御を受け取るたびに SHOW CALLS コマンドが実行され，その出力が CALLS に表示され，前の内容は消去されます。

```
DBG> DISPLAY CALLS AT Q3 DO (SHOW CALLS)
```

次のコマンドは，ベクタ・レジスタ V2 の要素 4 ～ 7 の内容を FORTRAN 配列の構文を使用して表示する V2_DISP という名前の DO ディスプレイを作成します。このディスプレイはデバッガが制御を受け取るたびに自動的に更新されます。

```
DBG> DISPLAY V2_DISP AT RQ2 DO (EXAMINE %V2(4:7))
```

DO ディスプレイに割り当てられるメモリ・バッファの省略時のサイズは 64 行です。メモリ・バッファが満杯になると，最も古い行が破棄され，次のテキストを表示するための空間が確保されます。バッファ・サイズを変更するには， DISPLAY/SIZE コマンドを使用します。

7.2.2 INSTRUCTION ディスプレイ対象

機械語命令ディスプレイは，ルーチンの命令ストリーム内にある EXAMINE /INSTRUCTION コマンドの出力を表示します。表示される命令はデバッグされているイメージからデコードされたものであり，実行されているコードを正確に示しているので，この種のディスプレイは最適化されたコードをデバッグするのに特に役立ちます (第 14.1 節を参照)。

このディスプレイでは 1 行が命令 1 つに相当します。命令に対応するソース行番号は左側の欄に表示されます。検査中の記憶位置にある命令はディスプレイの中央に置かれ，左側の欄の矢印によってマークされます。

機械語命令ディスプレイへ書き込みを行うには，前もって SELECT/INSTRUCTION コマンドでそのディスプレイを**現在の機械語命令ディスプレイ**として選択しておかなければなりません。

次の例では， DISPLAY コマンドで機械語命令ディスプレイ INST2 を RH1 に作成します。その後， SELECT/INSTRUCTION コマンドで INST2 を現在の機械語命令ディスプレイとして選択します。 EXAMINE/INSTRUCTION X コマンドを実行した時点で，ウィンドウ RH1 には X で表した記憶位置の前後にある命令ストリームが表示

されます。記憶位置 X はディスプレイの中央に置かれ，矢印がその位置の命令を指します。

```
DBG> DISPLAY INST2 AT RH1 INSTRUCTION
DBG> SELECT/INSTRUCTION INST2
DBG> EXAMINE/INSTRUCTION X
```

これ以後， EXAMINE/INSTRUCTION コマンドを実行するたびにディスプレイが更新されます。

命令ディスプレイに割り当てられるメモリ・バッファの省略時のサイズは 64 行です。しかし，前後にスクロールすれば，ルーチン内のすべての命令を表示できます。バッファ・サイズを変更して性能を向上するには， DISPLAY/SIZE コマンドを使用します。

7.2.3 INSTRUCTION(コマンド) ディスプレイ対象

これは，指定したコマンドの出力によって自動的に更新される機械語命令ディスプレイです。そのコマンド (EXAMINE/INSTRUCTION コマンドでなければならない) は，デバッガがユーザ・プログラムから制御を受け取るたびに実行されます。

たとえば，次のコマンドは機械語命令ディスプレイ INST3 をウィンドウ RS45 に作成します。デバッガが制御を受け取るたびに，組み込みコマンド EXAMINE /INSTRUCTION .%INST_SCOPE\%PC が実行され，ディスプレイが更新されます。

```
DBG> DISPLAY INST3 AT RS45 INSTRUCT (EX/INST .%INST_SCOPE\%PC)
```

このコマンドは，定義済みディスプレイ INST に似た機能を持つディスプレイを作成します。組み込み EXAMINE/INSTRUCTION コマンドは現在の有効範囲内で現在の PC 値にある命令を表示します (第 7.4.4 項を参照)。

自動的に更新される機械語命令ディスプレイを現在の機械語命令ディスプレイとして選択してある場合，そのディスプレイはその組み込みコマンドによって更新されるほかに，会話形式の EXAMINE/INSTRUCTION コマンドによって単純な機械語命令ディスプレイのように更新されます。

命令ディスプレイに割り当てられるメモリ・バッファの省略時のサイズは 64 行です。しかし，前後にスクロールすれば，ルーチン内のすべての命令を表示できます。バッファ・サイズを変更して性能を向上するには， DISPLAY/SIZE コマンドを使用します。

7.2.4 OUTPUT ディスプレイ対象

出力ディスプレイは別のディスプレイへ出力されないデバッガ出力を表示します。新しい出力は前のディスプレイ内容に追加されます。

出力ディスプレイへ書き込みを行うには，前もってそのディスプレイを SELECT /OUTPUT コマンドで**現在の出力ディスプレイ**として選択しておくか， SELECT /ERROR コマンドで**現在のエラー・ディスプレイ**として選択しておくか，あるいは SELECT/INPUT コマンドで**現在の入力ディスプレイ**として選択しておかなければなりません。出力ディスプレイに対する SELECT コマンドの使用についての詳しい説明は，第 7.3 節を参照してください。

次の例では， DISPLAY コマンドで出力ディスプレイ OUT2 をウィンドウ T2 に作成します (ディスプレイ対象 OUTPUT は省略時のディスプレイ対象なので，この例では削除してもかまいません)。その後， SELECT/OUTPUT コマンドで OUT2 を現在の出力ディスプレイとして選択します。この 2 つのコマンドによって，定義済みディスプレイ OUT に似た機能を持つディスプレイが作成されます。

```
DBG> DISPLAY OUT2 AT T2 OUTPUT
DBG> SELECT/OUTPUT OUT2
```

その結果， OUT2 は他のディスプレイへ出力されないデバッガ出力をすべて収集するようになります。次に例を示します。

- SHOW CALLS コマンドの出力は OUT2 へ出力される。
- 現在の機械語命令ディスプレイとして選択されている機械語命令ディスプレイがない場合， EXAMINE/INSTRUCTION コマンドの出力は OUT2 へ出力される。
- 省略時の設定では，デバッガの診断メッセージは PROMPT ディスプレイへ出力される。この出力は， SELECT/ERROR コマンドで OUT2 へ出力することができる。

出力ディスプレイに割り当てられるメモリ・バッファの省略時のサイズは 64 行です。メモリ・バッファが満杯になると，最も古い行が破棄され，次のテキストを表示するための空間が確保されます。バッファ・サイズを変更するには， DISPLAY/SIZE コマンドを使用します。

7.2.5 レジスタ・ディスプレイ対象

レジスタ・ディスプレイは自動的に更新されるディスプレイであり，プロセッサ・レジスタの現在の値が 16 進形式で表示され，また，表示できる数だけ呼び出しスタックの値も表示されます。

表示されるレジスタ値は，現在実行が中断されているルーチンの値です。デバッガに制御が渡されると，必ず値が更新されます。変更された値は強調表示されます。

事前定義のレジスタ・ディスプレイが，最大 3 個まであります。 REG ディスプレイ， IREG ディスプレイ，および FREG ディスプレイはが Alpha プロセッサ，および Itanium®プロセッサで事前定義されています。事前定義されているディスプレイの内容を，表 7–1 に示します。

表 7–1 事前定義のレジスタ・ディスプレイ

ディスプレイ	Alpha	Intel® Itanium®
REG	- R0 ～ R31 - PC - PS - F0 ～ F31 - FPCR - SFPCR - 呼び出しスタック値のトップ	- PC - CFM - R1 ～ R31 - R32 ～ R127 (使用されている分) - F2 ～ F127 - スタック値のトップ
IREG	- R0 ～ R31 - PC - PS - 呼び出しスタック値のトップ データは， 16 進形式で表示されます。	- PC - CFM - R1 ～ R31 - 呼び出しスタック値のトップ データは， 16 進形式で表示されます。
FREG	- F0 ～ F31 - FPCR - SFPCR - 呼び出しスタック値のトップ データは，浮動小数点形式で表示されます。	- F2 ～ F127 - スタック値のトップ レジスタ・データは， データ値 (整数または浮動小数点) に合った形式で表示されます。 スタック値は，浮動小数点形式で表示されます。

Alpha プロセッサでは，定義済みディスプレイ REG には，汎用レジスタ R0 ～ R28， FP (R29)， SP (R30)， R31， PC， PS，浮動小数点レジスタ F0 ～ F31， FPCR， SFPCR が 16 進形式で格納され，また，表示できる数だけ呼び出しスタックの値も格納されます。

Alpha プロセッサでは，定義済みディスプレイ IREG には，汎用レジスタ R0 ～ R28， FP，およびウィンドウに表示できる数だけ，呼び出しスタックの値が 16 進形式で格納されます。

Alpha プロセッサでは，定義済みディスプレイ FREG には，浮動小数点レジスタ F0 ～ F31， FPCR， SFPCR が浮動小数点形式で格納され，また，ウィンドウに表示できる数だけ，呼び出しスタックの値が 16 進形式で格納されます。

レジスタ・ディスプレイに割り当てられるメモリ・バッファの省略時のサイズは 64 行です。メモリ・バッファが満杯になると，最も古い行が破棄され，新しいテ

キストを格納するための空間が確保されます。バッファ・サイズを変更するには，DISPLAY/SIZE コマンドを使用します。

7.2.6 SOURCE ディスプレイ対象

ソース・ディスプレイは，あるモジュールのソース・コードが取得できる場合に，そのソース・コード内の TYPE コマンドまたは EXAMINE/SOURCE コマンドの出力を表示します。ソース行番号は左側の欄に表示されます。コマンドの出力であるソース行はディスプレイの中央に置かれ，左側の欄の矢印によって示されます。 TYPE コマンドで行の範囲を指定した場合は，それらの行がディスプレイの中央に置かれますが，矢印は表示されません。

ソース・ディスプレイへ書き込みを行うには，前もって SELECT/SOURCE コマンドでそのディスプレイを現在のソース・ディスプレイとして選択しておかなければなりません。

次の例では， DISPLAY コマンドでソース・ディスプレイ SRC2 を Q2 に作成します。その後， SELECT/SOURCE コマンドで SRC2 を現在のソース・ディスプレイとして選択します。 TYPE 34 コマンドを実行した時点で，ウィンドウ RH1 にはデバッグ中のモジュールの行 34 の前後にあるソース・コードが表示されます。矢印はディスプレイの中央に置かれた行 34 を指します。

```
DBG> DISPLAY SRC2 AT Q2 SOURCE
DBG> SELECT/SOURCE SRC2
DBG> TYPE 34
```

これ以後， TYPE コマンドまたは EXAMINE/SOURCE コマンドを実行するたびにディスプレイが更新されます。

ソース・ディスプレイのメモリ・バッファの省略時のサイズは 64 行です。ソース・ディスプレイのメモリ・バッファはページングされるため，ソース・モジュール全体またはルーチン全体で前後にスクロールします。バッファ・サイズを変更して性能を向上するには， DISPLAY/SIZE コマンドを使用します。

7.2.7 SOURCE (コマンド) ディスプレイ対象

これは，指定したコマンドの出力によって自動的に更新されるソース・ディスプレイです。そのコマンド (EXAMINE/SOURCE コマンドまたは TYPE コマンドでなければならない) は，デバッガがユーザ・プログラムから制御を受け取るたびに実行されます。

たとえば，次のコマンドはソース・ディスプレイ SRC3 をウィンドウ RS45 に作成します。デバッガが制御を受け取るたびに，組み込みコマンド EXAMINE/SOURCE .%SOURCE_SCOPE\%PC が実行され，ディスプレイが更新されます。

```
DBG> DISPLAY SRC3 AT RS45 SOURCE (EX/SOURCE .%SOURCE_SCOPE\%PC)
```

このコマンドは定義済みディスプレイ SRC に似た機能を持つディスプレイを作成します。組み込み EXAMINE/SOURCE コマンドは現在の有効範囲内で現在の PC 値に対するソース行を表示します (第 7.4.1 項を参照)。

自動的に更新されるソース・ディスプレイを現在のソース・ディスプレイとして選択してある場合，そのディスプレイはその組み込みコマンドによって更新されるほかに，会話形式の EXAMINE/SOURCE コマンドまたは TYPE コマンドによって単純なソース・ディスプレイのように更新されます。

ソース・ディスプレイのメモリ・バッファの省略時のサイズは 64 行です。自動的に更新されるソース・ディスプレイのメモリ・バッファはページングされるため，ソース・モジュール全体またはルーチン全体で前後にスクロールできます。バッファ・サイズを変更して性能を向上するには， DISPLAY/SIZE コマンドを使用します。

7.2.8 PROGRAM ディスプレイ対象

プログラム・ディスプレイには，デバッグされるプログラムの出力が格納されます。定義済み PROMPT ディスプレイはプログラム・ディスプレイに属し，この種類で使用できる唯一のディスプレイです。プログラム・ディスプレイとして新しいディスプレイを作成することはできません。

混乱を避けるため， PROMPT ディスプレイにはいくつかの制限事項があります (第 7.4.3 項を参照)。

7.3 ディスプレイ属性の割り当て

画面モードでは，ユーザが会話形式で入力したコマンドの出力は，その出力の種類および各種のディスプレイへ割り当てたディスプレイ属性に応じて各種のディスプレイに出力されます。たとえば，デバッガの診断メッセージは error 属性を持つディスプレイ (現在のエラー・ディスプレイ) へ出力されます。ディスプレイへ 1 つまたは複数の属性を割り当てることにより，異なる種類の情報を混合したり分離したりできます。

属性には次の名前が付いています。

error
input
instruction
output
program
prompt
scroll

source

ディスプレイに属性を割り当てた場合，その属性の名前がウィンドウ最上部の境界上に小文字でディスプレイ名の右側に表示されます。 scroll 属性はデバッガ出力に影響を及ぼすことはありませんが， SCROLL, MOVE, EXPAND の各コマンドの省略時のディスプレイを制御するために使用されます。

省略時の設定では，定義済みディスプレイに次のような属性が割り当てられます。

- SRC は source 属性と scroll 属性を持つ。
- OUT は output 属性を持つ。
- PROMPT は prompt, program，および error の各属性を持つ。

ディスプレイに属性を割り当てるには，属性と同じ名前の修飾子を指定した SELECT コマンドを使用します。次の例では， DISPLAY コマンドで出力ディスプレイ ZIP を作成します。その後， SELECT/OUTPUT コマンドで ZIP を現在の出力ディスプレイ (output 属性を持つディスプレイ) として選択します。後者のコマンドを実行した後， "output"という語は定義済み出力ディスプレイ OUT の最上部境界から消え，ディスプレイ ZIP 上に現れます。また，それ以前に OUT へ出力されていたすべてのデバッガ・ディスプレイが ZIP へ出力されるようになります。

```
DBG> DISPLAY ZIP OUTPUT
DBG> SELECT/OUTPUT ZIP
```

特定の属性は，ある決まったディスプレイ対象にだけ割り当てることができます。次のリストは SELECT コマンドの各修飾子とその効果，およびその属性を割り当てることができるディスプレイ対象を示したものです。

SELECT 修飾子	ディスプレイ対象への適応	説明
/ERROR	出力 プロンプト	指定した表示を現在のエラー・ディスプレイとして選択する。これ以後のデバッガの診断メッセージは，すべてそのディスプレイへ出力される。指定するディスプレイは出力ディスプレイか PROMPT ディスプレイでなければならない。ディスプレイを指定しなかった場合は，PROMPT ディスプレイが現在のエラー・ディスプレイとして選択される。
/INPUT	出力	指定したディスプレイを現在の入力ディスプレイとして選択する。これ以後のデバッガ入力は，すべてそのディスプレイ内にエコーバックされる。指定するディスプレイは出力ディスプレイでなければならない。ディスプレイを指定しなかった場合は現在の入力ディスプレイが選択解除され，デバッガ入力はどのディスプレイにもエコーバックされなくなる。

SELECT 修飾子	ディスプレイ対象への適応	説明
/INSTRUCTION	命令	指定したディスプレイを現在の機械語命令ディスプレイとして選択する。これ以後の EXAMINE /INSTRUCTION コマンドの出力は，すべてそのディスプレイへ出力される。指定するディスプレイは機械語命令ディスプレイでなければならない。キーパッド・キー・シーケンス PF4 COMMA は，ディスプレイ・リスト内の次の機械語命令ディスプレイを現在の機械語命令ディスプレイとして選択する。ディスプレイを指定しなかった場合は現在の機械語命令ディスプレイが選択解除され，どのディスプレイも instruction 属性を持たなくなる。
/OUTPUT	出力 プロンプト	指定したディスプレイを現在の出力ディスプレイとして選択する。これ以後のデバッガ出力は，すべてそのディスプレイへ出力される。ただし，特定の種類の出力が別のディスプレイへ出力される場合 (たとえば，現在のエラー・ディスプレイへ出力される診断メッセージなど) は除く。指定するディスプレイは出力ディスプレイか PROMPT ディスプレイでなければならない。キーパッド・キー・シーケンス PF1 KP3 は，ディスプレイ・リスト内の次の出力ディスプレイを現在の出力ディスプレイとして選択する。ディスプレイを指定しなかった場合は，PRO MPT ディスプレイが現在の出力ディスプレイとして選択される。
/PROGRAM	プロンプト	指定したディスプレイを現在のプログラム・ディスプレイとして選択する。これ以後のプログラム入出力 (I/O) は強制的にそのディスプレイで行われる。現在，PROM PT ディスプレイだけが指定できる。ディスプレイを指定しなかった場合は現在のプログラム・ディスプレイが選択解除され，プログラムの入出力が PROMPT ディスプレイへ強制的に送られることはなくなる。
/PROMPT	プロンプト	指定したディスプレイを現在のプロンプト・ディスプレイとして選択し，デバッガはそのディスプレイ内に入力用のプロンプトを表示する。現在，PROMPT ディスプレイだけが指定できる。PROMPT ディスプレイを選択解除することはできない。
/SCROLL	すべて	指定したディスプレイを現在のスクロール・ディスプレイとして選択する。そのディスプレイは，それ以後の SCROLL, MOVE, EXPAND の各コマンド用の省略時のディスプレイになる。ユーザはどのディスプレイでも指定できる。ただし，PROMPT ディスプレイはスクロールしない。SELECT コマンドで修飾子を指定しなかった場合は，この/SCROLL 修飾子が省略時の設定になる。キー KP3 は，ディスプレイ・リスト内で現在のスクロール・ディスプレイの後ろにある次のスクロール・ディスプレイを現在のスクロール・ディスプレイとして選択する。ディスプレイを指定しなかった場合は現在のスクロール・ディスプレイが選択解除され，どのディスプレイも scroll 属性を持たなくなる。

SELECT 修飾子	ディスプレイ対象への適応	説明
/SOURCE	ソース	指定したディスプレイを現在のソース・ディスプレイとして選択する。それ以後のすべての TYPE コマンドまたは EXAMINE/SOURCE コマンドの出力は，そのディスプレイへ出力される。指定するディスプレイはソース・ディスプレイでなければならない。キーパッド・キー・シーケンス PF4 KP3 は，ディスプレイ・リスト内の次のソース・ディスプレイを現在のソース・ディスプレイとして選択する。ディスプレイを指定しなかった場合は現在のソース・ディスプレイが選択解除され，どのディスプレイも source 属性を持たなくなる。

上記の表に示した制限事項に従って，ディスプレイはいくつかの属性を持つことができます。前に述べた例では， ZIP は現在の出力ディスプレイとして選択されました。次の例では， ZIP をさらに現在の入力ディスプレイ，エラー・ディスプレイ，およびスクロール・ディスプレイとして選択します。これらのコマンドの実行後は，デバッガの入出力 (I/O) と診断は発生した順に正しく ZIP 内に記録され， ZIP が現在のスクロール・ディスプレイになります。

```
DBG> SELECT/INPUT/ERROR/SCROLL ZIP
```

ディスプレイ属性のそれぞれに対して現在選択されているディスプレイを示すには，SHOW SELECT コマンドを使用します。

SELECT コマンドに特定の修飾子を指定し，ディスプレイ名を指定しなかった場合は，通常，その属性の割り当てが解除されます (その属性を持っていたディスプレイが選択解除されます)。その場合の正確な結果は，上記の表で説明したとおり属性によって異なります。

7.4 定義済みディスプレイ

デバッガは次の定義済みディスプレイを備えており，これらは各種のデータを取り込んで表示するために使用できます。

SRC (定義済みソース・ディスプレイ)
OUT (定義済み出力ディスプレイ)
PROMPT (定義済みプロンプト・ディスプレイ)
INST (定義済み機械語命令ディスプレイ)
REG (定義済みレジスタ・ディスプレイ)
FREG (定義済み浮動小数点レジスタ・ディスプレイ (Alpha のみ))
IREG (定義済み整数レジスタ・ディスプレイ)

画面モードに入ると，デバッガは図 7–1 に示したように SRC を画面の上半分に，PROMPT を画面下部の 6 分の 1 に，そして OUT を SRC と PROMPT の間に配置します。省略時の設定では， INST， REG， FREG (Alpha のみ) および IREG のディスプレイは初期の画面から除去されています。

ディスプレイとウィンドウを並べ替えたあとに，この省略時の構成を再作成するには，キーパッドの BLUE MINUS を押します (PF4 のあとで MINUS(–)キーを押します)。

定義済みディスプレイの基本的な機能は次の各項で説明します。ユーザはこれらのディスプレイのバッファ・サイズやディスプレイ属性など，特定の特性を変更することができます。また，定義済みディスプレイに似た追加のディスプレイを作成することもできます。

ディスプレイの特性に関する要約情報を表示するには， SHOW DISPLAY コマンドを使用します。

表 7–2 は，定義済みディスプレイに関する重要な情報をまとめています。

表 7–2 定義済みディスプレイ

ディスプレイ名	ディスプレイ対象	有効なディスプレイ属性	起動時に表示されるかどうか
SRC	ソース	スクロール ソース (省略時の設定)	X
OUT	出力	エラー 入力 出力 (省略時の設定) スクロール	X
PROMPT	出力	エラー (省略時の設定) 出力 プログラム (省略時の設定) プロンプト (省略時の設定) スクロール[1]	X
INST	命令	命令 スクロール	
REG	レジスタ	スクロール	
FREG (Alpha のみ)	レジスタ	スクロール	
IREG	レジスタ	スクロール	

[1]定義済み PROMPT ディスプレイはスクロールできない。

7.4.1 定義済みソース・ディスプレイ (SRC)

注意

デバッグ・セッション中にソース・コードをディスプレイに使用する方法についての詳しい説明は，第 6 章を参照してください。

定義済みディスプレイの SRC (図 7–1 を参照) は，自動的に更新されるソース・ディスプレイです。

SRC は，ソース・コードを次の 2 つの基本的な方法で表示するために使用できます。

- 省略時の設定では， SRC は現在実行が一時停止しているモジュールのソース・コードを自動的に表示する。これによって，ユーザは現在のデバッグ・コンテキストを素早く判別できる。
- 省略時の設定では SRC は source 属性を持っているので，第 7.4.1.1 項で説明するとおり， SRC を使用してプログラムの任意の部分のソース・コードを表示できる。

ソース・コードが表示されているモジュールの名前はディスプレイ名 SRC の右側に示されます。ソース・コードの左側に表示される番号は，コンパイラ生成リスト・ファイルに現れるコンパイラ生成行番号です。

デバッガの制御下でプログラムを実行すると，実行が一時停止するたびに SRC が自動的に更新されます。左端の欄の矢印は，次に実行されるソース行を指しています。正確に言えば，実行はそのソース行に対応する最初の命令の位置で一時停止しています。したがって，矢印が指している行は現在のプログラム・カウンタ (PC) の値に対応します。 PC は，次に実行する命令のメモリ・アドレスが入っているレジスタです。

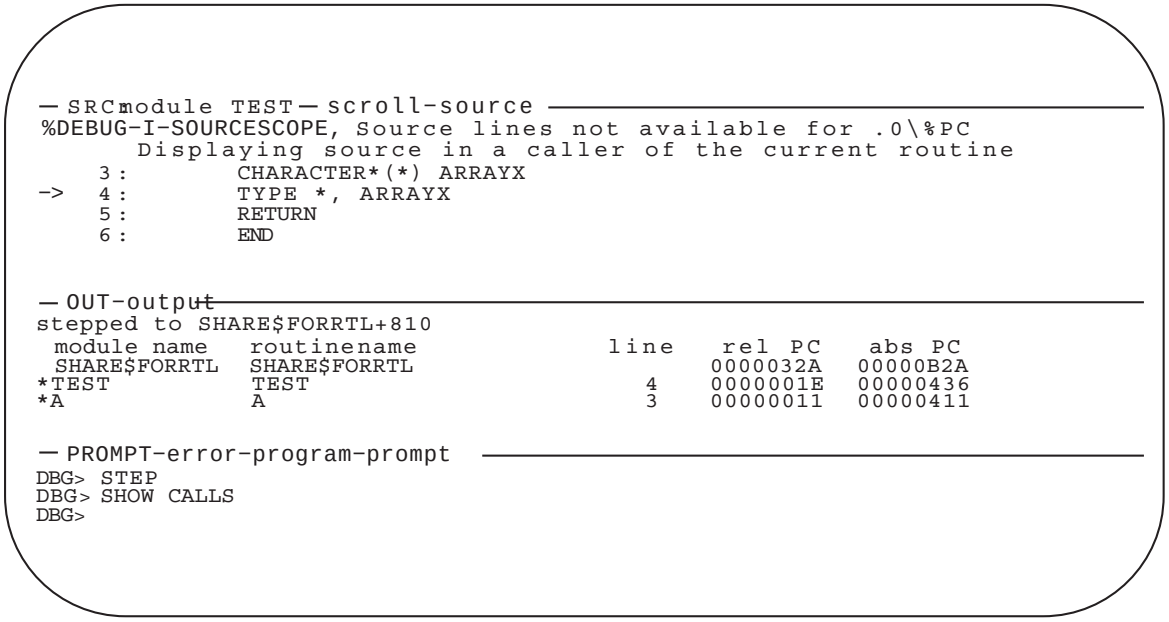
実行が一時停止しているルーチンのソース・コードを検索できない場合，たとえば，そのルーチンが実行時ライブラリ・ルーチンである場合などは，デバッガは呼び出しスタック上で次に下にあるルーチン (ソース・コードが取得できるもの) のソース・コードを表示しようとします。そのようなルーチンのソース・コードを表示する場合，デバッガは次のメッセージを発行します。

```
%DEBUG-I-SOURCESCOPE, Source lines not available for .0\%PC.
        Displaying source in a caller of the current routine.
```

図 7–2 はこの機能を示しています。ソース・ディスプレイには，ルーチン TYPE の呼び出しが現在アクティブであることが示されています。 TYPE は Fortran 実行時ライブラリ・プロシージャの 1 つに対応します。このルーチンのソース・コードは取得できないので，デバッガは呼び出し元ルーチンのソース・コードを表示します。出力ディスプレイに示されている SHOW CALLS コマンドの出力は，実行が一時停止しているルーチンとそのルーチンに通じる呼び出しシーケンスを示しています。

このような場合，ソース・ウィンドウの矢印はそのルーチン呼び出し後に実行が戻ってくる行を示します。この行は，ソース言語とコーディング・スタイルに応じて，その呼び出し文を含んでいる行であったり，その次の行であったりします。

図 7–2 ソース・コードが取得できない場合の画面モード・ソースの表示



ZK–6504–GE

プログラムがコンパイル時に最適化された場合は，SRC に表示されるソース・コードが実際に実行されているコードを表さないこともあります。そのような場合には，定義済み機械語命令ディスプレイの INST が役立ちます。このディスプレイは実行されている命令を正確に示します。(第 7.4.4 項を参照。)

ディスプレイ SRC を自動的に更新する組み込みコマンドはEXAMINE/SOURCE .%SOURCE_SCOPE\%PCです。 EXAMINE/SOURCE コマンドについての詳しい説明は第 6.4 節を参照してください。組み込みデバッガ・シンボル%SOURCE_SCOPE は 1 つの有効範囲を表し，次の特性を持っています。

- 省略時の設定では，%SOURCE_SCOPE は有効範囲 0 を表します。これは実行が現在一時停止しているルーチンの有効範囲です。
- SET SCOPE/CURRENT コマンドを使用して有効範囲検索リストを呼び出しスタックの相対位置に再設定した場合 (第 7.4.1.2 項を参照)，%SOURCE_SCOPE は指定された現在の有効範囲 (検索リストの始めのルーチンの有効範囲) を表します。
- 現在の有効範囲にあるルーチンのソース・コードが取得できない場合，%SOURCE_SCOPE は有効範囲*n*を表します。ここで，*n*は呼び出しスタックの下にあってソース・コードが取得できる最初のレベルです。

7.4.1.1 任意のプログラム記憶位置でのソース・コードの表示

プログラム全体を通じて，ソース・コードがディスプレイに使用できる場合はソース・コードを表示するためにディスプレイ SRC を使用できます。

- 第 7.5.1 項で説明するとおり， KP2(下スクロール) または KP8(上スクロール) を押すことによって，ソース・ディスプレイ全体にわたってスクロールすることができる。この結果，実行が一時停止しているモジュールのどのソース・コードでも参照できる。
- SET SCOPE/CURRENT コマンド (第 7.4.1.2 項を参照) を使用すれば，現在呼び出しスタック上にあるどのルーチンのソース・コードでも表示できる。
- SRC は source 属性を持っているので，プログラム全体を通して， TYPE コマンドおよび EXAMINE/SOURCE コマンドを使用することによってソース・コードを表示できる。
  - 任意のソース行を表示するには， TYPE コマンドを使用する (第 6.3 節を参照)。
  - あるコード記憶位置 (たとえば，ルーチン宣言) に対応したソース行を表示するには， EXAMINE/SOURCE コマンドを使用する (第 6.4 節を参照)。

  TYPE コマンドまたは EXAMINE/SOURCE コマンドを使用する場合，ソース・コードを参照したいモジュールがまず設定されていることを確かめる必要がある。特定のモジュールが設定されているかどうかを判別するには， SHOW MODULE コマンドを使用する。その後，必要であれば SET MODULE コマンドを使用する (第 5.2 節を参照)。

ディスプレイ SRC の内容の操作後， KP5 を押せば現在実行が一時停止している記憶位置を再表示できます (SRC の省略時の動作)。

7.4.1.2 呼び出しスタック上にあるルーチンのソース・コードの表示

SET SCOPE/CURRENT コマンドを使用すると，現在呼び出しスタック上にあるルーチンのソース・コードを表示できます。たとえば次のコマンドは，現在実行が一時停止しているルーチンの呼び出し元のソース・コードが表示されるようにディスプレイ SRC を更新します。

```
DBG> SET SCOPE/CURRENT 1
```

ソース・コードを表示するための省略時の有効範囲を再設定するには， CANCEL SCOPE コマンドを入力します。このコマンドは，実行が一時停止している呼び出しスタック最上部のルーチンのソース・コードがディスプレイ SRC に表示されるようにします。

7.4.2 定義済み出力ディスプレイ (OUT)

図 7–1 および図 7–2 は，定義済みディスプレイ OUT の代表的なデバッガ出力を示しています。

ディスプレイ OUT は汎用の出力ディスプレイです。省略時の設定では OUT は output 属性を持っているので，ソース・ディスプレイ SRC や機械語命令ディスプレイ INST には出力されないデバッガ出力を表示します。たとえば，ディスプレイ INST を更新するはずの出力は，ディスプレイ INST が表示されていないか instruction 属性を持っていなければディスプレイ OUT の中に表示されます。

省略時の設定では， OUT はデバッガの診断メッセージを表示せず，それらは PROMPT ディスプレイに表示されます。ユーザは OUT に属性を割り当てることによって，通常の出力だけでなくデバッガの入力と診断も OUT に取り込まれるようにすることができます (第 7.3 節を参照)。

省略時の設定では，定義済みディスプレイ OUT に割り当てられるメモリ・バッファは 100 行です。

7.4.3 定義済みプロンプト・ディスプレイ (PROMPT)

定義済みディスプレイ PROMPT は，デバッガが入力を求めるプロンプトを出すディスプレイです。図 7–1 および図 7–2 では， PROMPT がその省略時の位置，つまり画面の下部の 6 分の 1 に示されています。

省略時の設定では， PROMPT はプロンプト属性を持ちます。さらに， PROMPT には (省略時の設定により) プログラム属性とエラー属性があり，プログラムからの出力と診断メッセージはそのディスプレイに出力されます。

PROMPT には，次に示すように他のディスプレイと異なる特性と制限事項があります。これは PROMPT ディスプレイを操作する際に混乱が起きないようにするためです。

- PROMPT ディスプレイ・ウィンドウはつねにすべて見える状態になっています。 PROMPT を表示しなかったり (DISPLAY/HIDE コマンド)， PROMPT をペーストボードから除去したり (DISPLAY/REMOVE コマンド)， PROMPT を削除したり (CANCEL DISPLAY コマンド) することはできません。
- PROMPT にスクロール属性を割り当てると， MOVE コマンドと EXPAND コマンドの出力がこのディスプレイに渡されます。しかし， PROMPT ディスプレイをスクロールすることはできません。
- PROMPT ディスプレイ・ウィンドウは常に最初のカラムから画面の幅いっぱいを占めます。
- PROMPT は，画面の任意の場所へ移動したり，全画面を覆う高さまで拡大したり， 2 行にまで縮小したりできます。

PROMPT によって隠されるようなディスプレイの移動または拡大をユーザが行おうとした場合，デバッガは警報を出します。

7.4.4 定義済み機械語命令ディスプレイ (INST)

注意

省略時の設定では，定義済み機械語命令ディスプレイ INST は画面に表示されず，命令属性も持ちません (第 7.4.4.1 項および第 7.4.4.2 項を参照)。

ディスプレイ INST は自動的に更新される機械語命令ディスプレイです。そのディスプレイはユーザ・プログラムのデコード済み命令ストリームを表示します。これは実行されている正確なコードであり，コンパイラの最適化の効果も含まれています。

VAX の例を図 7–3 に示します。

このディスプレイは最適化されたコードのデバッグに便利です。最適化されたコードの場合，実行されているコードがソース・ディスプレイに表示されているコードと一致しないこともあります。最適化の効果についての詳しい説明は第 14.1 節を参照してください。

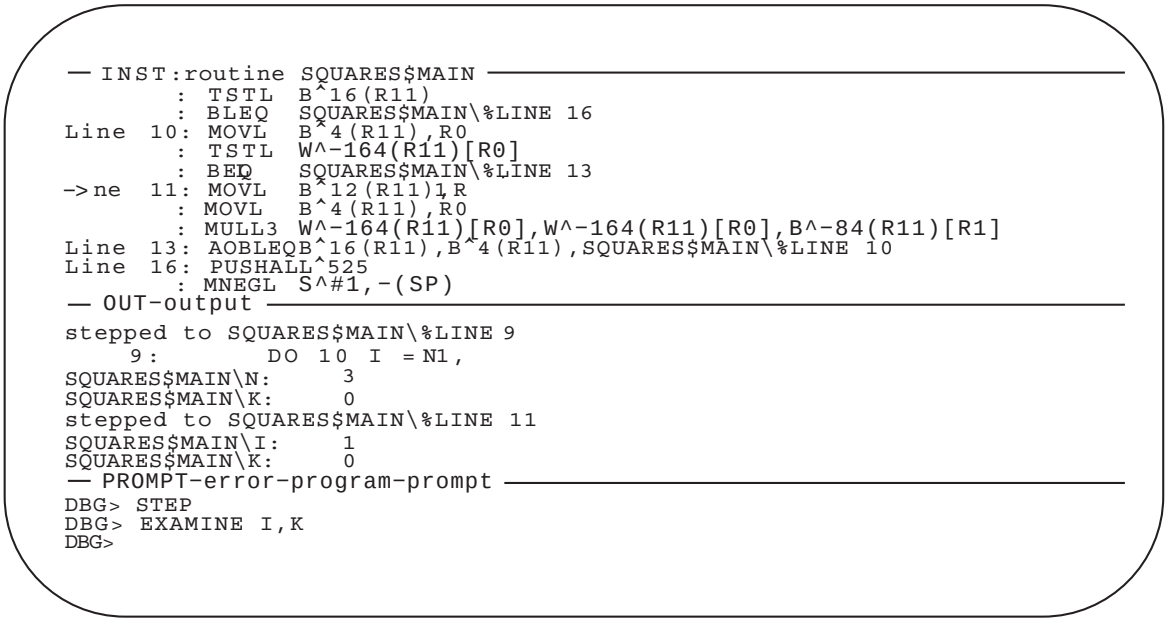
INST は，次の 2 つの基本的な方法で使用できます。

- 省略時の設定では， INST は現在実行が一時停止しているルーチンのデコード済み命令を表示する。その結果，現在のデバッグ・コンテキストを素早く判別できる。
- INST が命令属性を持っている場合，第 7.4.4.2 項で説明するとおり，プログラムの任意の部分のデコード済み命令を表示するために INST を使用することができる。

命令が表示されているルーチンの名前は，ディスプレイ名「INST」の右側に表示されます。命令の左側に表示される番号はコンパイラ生成ソース行番号です。

デバッガの制御下でプログラムを実行している場合は，実行が一時停止するたびに自動的に INST が更新されます。左端の欄の矢印は実行が一時停止している位置の命令を指します。これは次に実行される予定の命令であり，この命令のアドレスが現在の PC 値です。

図 7–3 画面モード機械語命令ディスプレイ (VAX システムの例)



ZK–6505–GE

ディスプレイ INST を自動的に更新する組み込みコマンドは EXAMINE/INSTRUCTION .%INST_SCOPE\%PCです。 EXAMINE /INSTRUCTION コマンドについての詳しい説明は，第 4.3.1 項を参照してください。組み込みデバッガ・シンボル%INST_SCOPE は有効範囲を表し，次の特性を持っています。

- 省略時の設定では，%INST_SCOPE は有効範囲 0 を表す。これは実行が現在一時停止しているルーチンの有効範囲である。
- SET SCOPE/CURRENT コマンドを使用して有効範囲検索リストを呼び出しスタックの相対位置に再設定した場合 (第 7.4.4.3 項を参照)，%INST_SCOPE は指定された現在の有効範囲 (検索リストの始めのルーチンの有効範囲) を表す。

7.4.4.1 機械語命令ディスプレイの表示

省略時の設定では，ディスプレイ INST はディスプレイ・ペーストボードから除去されたものとしてマークされ (第 7.5.2 項を参照)，可視ではありません。ディスプレイ INST を表示するには，次のいずれかの方法を使用します。

- ディスプレイ SRC とディスプレイ INST を隣り合わせに配置するには KP7 を押す。この結果，ソース・コードとデコード済み命令ストリームが比較できる。
- ディスプレイ INST とディスプレイ REG を隣り合わせに配置するには PF1 KP7 を押す。
- INST を省略時の位置または最も新しく定義された位置に配置するには DISPLAY INST コマンドを入力する (第 7.5.2 項を参照)。

7.4.4.2 任意のプログラム記憶位置にある命令の表示

INST は，プログラム全体を通してデコード済み命令を表示するために使用することができます。

- 第 7.5.1 項で説明するとおり， KP2(下スクロール) または KP8(上スクロール) を押すことによって，機械語命令ディスプレイ全体にわたってスクロールすることができる。この結果，実行が一時停止しているルーチン内のどの命令でも参照できる。
- SET SCOPE/CURRENT コマンド (第 7.4.4.3 項を参照) を使用すれば，現在呼び出しスタック上にあるどのルーチンの命令ストリームでも表示できる。
- INST が instruction 属性を持っている場合は，プログラム全体を通して，EXAMINE/INSTRUCTION コマンドを使用することによって任意のコード記憶位置の命令を次のように表示できる。
  - INST に instruction 属性を割り当てるには， SELECT/INSTRUCTION INST コマンドを使用する (第 7.2.2 項および第 7.3 節を参照)。 KP7 または PF1 KP7 を押して INST を表示した場合は，自動的に INST に instruction 属性が割り当てられる。
  - コード記憶位置 (たとえば，ルーチンの宣言) に対応した命令を表示するには，EXAMINE/INSTRUCTION コマンド (第 4.3.1 項を参照) を使用する。

  instruction 属性を持っている表示がない場合， EXAMINE/INSTRUCTION コマンドの出力はディスプレイ OUT へ出力されます。

ディスプレイ INST の内容の操作後， KP5 を押せば現在実行が一時停止している記憶位置を再表示 (INST の省略時の動作) できます。

7.4.4.3 呼び出しスタック上にあるルーチンの命令の表示

SET SCOPE/CURRENT コマンドを使用すると，現在呼び出しスタック上にあるルーチンの命令を表示できます。たとえば，次のコマンドはディスプレイ INST を更新し，現在実行が一時停止しているルーチンの呼び出し元の命令をディスプレイ INST に表示します。

```
DBG> SET SCOPE/CURRENT 1
```

命令を表示するための省略時の有効範囲を再設定するには， CANCEL SCOPE コマンドを入力します。このコマンドを入力すると，実行が一時停止されている呼び出しスタック最上部にあるルーチンの命令が表示されます。

7.4.4.4 呼び出しスタック上にあるルーチンのレジスタ値の表示

SET SCOPE/CURRENT コマンドを使用すると，現在呼び出しスタック上にあるルーチンに対応したレジスタ値を表示できます。たとえば，次のコマンドはディスプレイ REG を更新し，現在実行が一時停止しているルーチンの呼び出し元のレジスタ値をディスプレイ REG に表示します。

```
DBG> SET SCOPE/CURRENT 1
```

レジスタ値を表示するための省略時の有効範囲を再設定するには， CANCEL SCOPE コマンドを入力します。このコマンドを入力すると，実行が一時停止している呼び出しスタック最上部のルーチンのレジスタ値がディスプレイ REG に表示されます。

7.5 既存のディスプレイの操作

この節では，次の機能の実行方法を説明します。

- SELECT コマンドおよび SCROLL コマンドを使用したディスプレイのスクロール
- DISPLAY コマンドを使用したディスプレイの表示，非表示，除去， CANCEL DISPLAY コマンドを使用したディスプレイの永久削除，および SHOW DISPLAY コマンドを使用した既存のディスプレイの識別とそれらのディスプレイ・リスト内の順序の識別
- MOVE コマンドを使用したディスプレイの画面内移動
- EXPAND コマンドを使用したディスプレイの拡大または縮小

第 7.7 節および第 7.2 節でも既存のディスプレイを DISPLAY コマンドによって変更するための方法 (ディスプレイ・ウィンドウおよび表示される情報の種類を変更する方法) を詳しく説明しています。

7.5.1 ディスプレイのスクロール

通常，ディスプレイはウィンドウを通して見ることができる以上の行数あるいは，ウィンドウより長い行のテキストを持っています。 SCROLL コマンドを使用すると，ウィンドウの境界を越えて隠れているテキストを見ることができます。 PROMPT ディスプレイを除くすべてのディスプレイでスクロールが可能です。

ディスプレイをスクロールさせる最も簡単な方法は，この節の後半で述べるようなキーパッド・キーを使用した方法です。最初に関連コマンドの使用法を説明します。

SCROLL コマンドではディスプレイを明示的に指定できます。しかし，通常はまず現在のスクロール・ディスプレイを選択するために SELECT/SCROLL コマンドを使用します。この結果，そのディスプレイは scroll 属性を持ち， SCROLL コマンドの

省略時のディスプレイとなります。その後，そのディスプレイを指定した行数だけ上下に，あるいは指定した桁数だけ左右にスクロールさせるために， SCROLL コマンドをパラメータなしで使用します。スクロールする方向と距離はコマンド修飾子 (/UP:*n*, /RIGHT:*n*など) で指定します。

次の例では， SELECT コマンドでディスプレイ OUT を現在のスクロール・ディスプレイとして選択してから (/SCROLL は省略時の修飾子なので省略可能)， SCROLL コマンドによって下 18 行のテキストが表示されるように OUT をスクロールさせています。

```
DBG> SELECT OUT
DBG> SCROLL/DOWN:18
```

いくつかの便利な SELECT コマンド行と SCROLL コマンド行が次のようにキーパッド・キーに割り当てられています。キーパッド図については，付録 A を参照してください。

- KP3 を押すと，ディスプレイ・リスト内で現在のスクロール・ディスプレイの後ろにある次のディスプレイへ scroll 属性が割り当てられる。あるディスプレイを現在のスクロール・ディスプレイとして選択したい場合は，そのディスプレイの最上行に"scroll"という語が現れるまで KP3 を繰り返し押す。
- 上下左右へスクロールさせるには，それぞれ KP8, KP2, KP6， KP4 を押す。スクロールする量は使用しているキー状態 (DEFAULT, GOLD，または BLUE) によって決まる。

7.5.2 ディスプレイの表示，非表示，除去，取り消し

DISPLAY コマンドは，ディスプレイを作成および操作するための最も用途の広いコマンドです。このコマンドを最も単純な形式で使用した場合，既存のディスプレイの 1 つがペーストボードの最上部に置かれ，そのディスプレイの現在のウィンドウを通して表示されます。たとえば，次のコマンドはディスプレイ INST をその現在のウィンドウを通して表示します。

```
DBG> DISPLAY INST
```

DISPLAY %NEXTDISP コマンドにバインドされている KP9 を押すと，同じことが簡単に行えます。組み込み関数%NEXTDISP はディスプレイ・リスト内の次のディスプレイを表します。付録 B に，画面に関連したすべての組み込み関数が示されています。 KP9 を押すたびに，リスト内の次のディスプレイがペーストボード最上部に置かれ，その現在のウィンドウ内に表示されます。

省略時の設定では，ディスプレイ OUT の最上行 (そのディスプレイを示す) はディスプレイ SRC の最下行に一致します。 SRC がペーストボード最上部にある場合，その最下行は OUT の最上行を隠します。各種のディスプレイをペーストボード最上部

に置くために DISPLAY コマンドとそれに対応するキーパッド・キーを使用する場合は，このことに注意してください。

ペーストボードの最下部にあるディスプレイを表示しないためには， DISPLAY /HIDE コマンドを使用します。このコマンドはディスプレイ・リスト内でのそのディスプレイの順番を変更します。

ペーストボードからディスプレイを除去して見えなくするには (ただし，永久に削除されるわけではない)， DISPLAY/REMOVE コマンドを使用します。除去したディスプレイをペーストボード上に戻すには， DISPLAY コマンドを使用します。

永久にディスプレイを削除するには， CANCEL DISPLAY コマンドを使用します。ディスプレイを再作成するには，第 7.6 節で説明するとおり， DISPLAY コマンドを使用します。

PROMPT ディスプレイは表示しなかったり，除去したり，削除したりできないので注意してください。

現在存在しているディスプレイを見るには， SHOW DISPLAY コマンドを使用します。ディスプレイはディスプレイ・リスト上の順序に従ってリストされます。ペーストボードの最上部にあるディスプレイは最後にリストされます。

DISPLAY オプションについての詳しい説明は，オンライン・ヘルプの DISPLAY コマンドの説明を参照してください。また， DISPLAY コマンドでは既存のディスプレイのその他の特性 (ディスプレイ・ウィンドウと表示される情報の種類) を変更できるオプション・パラメータが使用できることを覚えておいてください。これらの方法については，第 7.7 節および第 7.2 節を参照してください。

7.5.3 ディスプレイの画面内移動

画面内でディスプレイを移動するには， MOVE コマンドを使用します。修飾子 /UP:*n*, /DOWN:*n*, /RIGHT:*n*，および/LEFT:*n*は，ディスプレイを移動する方向と行数または桁数を指定します。ディスプレイを指定しなかった場合には現在のスクロール・ディスプレイが移動します。

ディスプレイを移動する最も簡単な方法は，次のようなキーパッド・キーを使用した方法です。

- 現在のスクロール・ディスプレイを選択するため， KP3 を必要なだけ繰り返し押す。
- キーパッドを MOVE 状態にして，ディスプレイを上下左右に移動するためにそれぞれ KP8, KP2,KP4, KP6 を押す。付録 A を参照。

7.5.4 ディスプレイの拡大または縮小

ディスプレイを拡大または縮小するには EXPAND コマンドを使用します。修飾子/UP:*n*, /DOWN:*n*, /RIGHT:*n*，および/LEFT:*n*は，ディスプレイを拡大または縮小する方向と行数または桁数を指定します。ディスプレイを縮小するには，これらの修飾子に負の整数値を指定します。ディスプレイを指定しなかった場合には現在のスクロール・ディスプレイが拡大または縮小されます。

ディスプレイを拡大または縮小する最も簡単な方法は，次のようなキーパッド・キーを使用した方法です。

- 現在のスクロール・ディスプレイを選択するため， KP3 を必要なだけ繰り返し押す。
- キーパッドを EXPAND 状態または CONTRACT 状態にしてから，ディスプレイを垂直方向または水平方向に拡大または縮小するために KP8, KP2, KP4，または KP6 を押す。付録 A を参照。

PROMPT ディスプレイは水平方向には縮小，拡大できません。また，垂直方向の場合， 2 行より少ない行数に縮小することはできません。

7.6 新しいディスプレイの作成

新しい画面ディスプレイを作成するには， DISPLAY コマンドを使用します。基本的な構文は次のとおりです。

```
DISPLAY display-name [AT window-spec] [display-kind]
```

ディスプレイ名は，すでに何かのディスプレイに付けてある名前以外であればどんな名前でもかまいません。既存のすべてのディスプレイを表示するには SHOW DISPLAY コマンドを使用します。新しいディスプレイを作成すると，そのディスプレイはペーストボードの最上部で，既存のディスプレイ (非表示にはできない定義済み PROMPT ディスプレイは除く) の最上部に置かれます。ディスプレイ名はディスプレイ・ウィンドウの左上隅にディスプレイされます。

第 7.7 節では，ウィンドウを指定するためのオプションを説明します。ウィンドウ指定を提供しなかった場合，そのディスプレイは画面の上半分または下半分に配置されます。この位置は新しいディスプレイを作成するたびに交互に変わります。

第 7.2 節では，ディスプレイ対象を指定するためのオプションを説明します。ディスプレイ対象を指定しなかった場合は出力ディスプレイが作成されます。

たとえば，次のコマンドは OUT2 という名前の新しい出力ディスプレイを作成します。 OUT2 に対応するウィンドウは画面の上半分または下半分となります。

```
DBG> DISPLAY OUT2
```

次のコマンドは EXAM_XY という名前の新しい DO ディスプレイを作成し，画面右の 3 番目の 4 分の 1(RQ3) に置きます。このディスプレイは変数 X と変数 Y の現在の値を示し，デバッガがプログラムから制御を受け取るたびに更新されます。

```
DBG> DISPLAY EXAM_XY AT RQ3 DO (EXAMINE X,Y)
```

詳しい説明については， DISPLAY コマンドの説明を参照してください。

7.7 ディスプレイ・ウィンドウの指定

ディスプレイ・ウィンドウは画面の任意の長方形部分を占めることができます。

ディスプレイ・ウィンドウは， DISPLAY コマンドでディスプレイを作成するときに指定できます。また， DISPLAY コマンドで新しいウィンドウを指定することによって，ディスプレイに現在対応するウィンドウを変更することもできます。ウィンドウを指定する場合，次のオプションがあります。

- 行数と桁数によってウィンドウを指定する。
- 定義済みウィンドウの名前，たとえば H1 などを使用する。
- 前もって SET WINDOW コマンドで設定したウィンドウ定義の名前を使用する。

これらの各方法については次の各項で説明します。ウィンドウを指定する場合，PROMPT ディスプレイが常にディスプレイ・ペーストボードの最上部に残り，したがって同じ画面領域を共有する別のディスプレイは一部が隠されることに注意してください。

ディスプレイ・ウィンドウは，指定した方法に関係なく動的なものです。したがって， SET TERMINAL コマンドを使用して画面の高さまたは幅を変更した場合，ディスプレイに対応したウィンドウは新しい画面の高さまたは幅に比例して拡大または縮小されます。

7.7.1 行数と桁数によるウィンドウの指定

ウィンドウ指定の一般的な形式は，(*start-line*,*line-count*[,*start-column*,*column-count*]) です。たとえば，次のコマンドは出力ディスプレイ CALLS を作成し，そのウィンドウは高さが 7 行で 10 行目から始まり，幅が 30 桁で 50 桁目から始まることを指定します。

```
DBG> DISPLAY CALLS AT (10,7,50,30)
```

*start-column*も*column-count*も指定しなかった場合，ウィンドウは画面いっぱいの幅になります。

7.7.2 定義済みウィンドウの使用

デバッガは多数の定義済みウィンドウを備えています。これらのウィンドウには，行数と桁数を指定しなくても DISPLAY コマンドで使用できる短いシンボリック名が付いています。たとえば，次のコマンドは出力ディスプレイ ZIP を作成し，そのウィンドウを RH1 (画面の右上半分) とすることを指定します。

```
DBG> DISPLAY ZIP AT RH1
```

SHOW WINDOW コマンドは，ユーザが SET WINDOW コマンドで作成したウィンドウ定義だけでなく，すべての定義済みウィンドウ定義も表示します。

7.7.3 新しいウィンドウ定義の作成

ほとんどの場合，定義済みウィンドウだけでも十分ですが， SET WINDOW コマンドで新しいウィンドウ定義を作成することもできます。このコマンドは次の構文をとり，ウィンドウ名とウィンドウ指定とを対応づけます。

```
SET WINDOW window-name AT (start-line,line-count[,
start-column,column-count])
```

ウィンドウ定義の作成後は， DISPLAY コマンドの中に定義済みウィンドウと同様にその名前を使用するだけです。次の例ではウィンドウ定義 MIDDLE が設定されます。その後，この定義を使用してディスプレイ OUT をウィンドウ MIDDLE を通して表示します。

```
DBG> SET WINDOW MIDDLE AT (9,4,30,20)
DBG> DISPLAY OUT AT MIDDLE
```

現在のすべてのウィンドウ定義を表示するには， SHOW WINDOW コマンドを使用します。ウィンドウ定義を削除するには， CANCEL WINDOW コマンドを使用します。

7.8 ディスプレイ構成のサンプル

画面モードの使用法は，ユーザ個人のニーズや見つけたいエラーの種類によって異なります。定義済みのディスプレイを使用するだけで十分な場合もあります。大きな画面へアクセスできるときには，追加の表示をさまざまな用途に作成したい場合があります。次の例をそのような場合の参考にしてください。

高級言語でデバッグを行っているときに，複数のルーチン呼び出しを通してプログラムの実行をトレースしたいと想定します。

まず，省略時の画面構成，つまり H1 に SRC， S45 に OUT， S6 に PROMPT を設定します。キーパッド・キー・シーケンス PF4 MINUS でこの構成が得られます。SRC には実行が一時停止しているモジュールのソース・コードが表示されます。

次のコマンドは，有効範囲 1 での PC 値 (実行が一時停止しているルーチンの呼び出しでの，呼び出しスタックの 1 レベル下) を示す SRC2 という名前のソース・ディスプレイを RH1 に作成します。

```
DBG> DISPLAY SRC2 AT RH1 SOURCE (EXAMINE/SOURCE .1\%PC)
```

この結果，画面の左半分には現在実行中のルーチンが表示され，右半分にはそのルーチンを呼び出し元が表示されます。

次のコマンドは，デバッガがプログラムから制御を受け取るたびに SHOW CALLS コマンドを実行する CALLS という名前の DO ディスプレイを S4 に作成します。

```
DBG> DISPLAY CALLS AT S4 DO (SHOW CALLS)
```

これによって OUT の上半分が CALLS で隠されるので，次のように OUT のウィンドウを小さくします。

```
DBG> DISPLAY OUT AT S5
```

同様なディスプレイ構成は，ソース・ディスプレイでなく機械語命令ディスプレイでも作成できます。

7.9 ディスプレイと画面状態の保存

SAVE コマンドを使用すると，既存のディスプレイのスナップショットを作成してそのコピーを新しいディスプレイとして保存することができます。これは，たとえば自動的に更新されるディスプレイ (DO ディスプレイなど) の現在の内容をあとで参照したい場合などに便利です。

次の例では， SAVE コマンドでディスプレイ CALLS の現在の内容をディスプレイ CALLS4 の中へ保存します。ディスプレイ CALLS4 は，このコマンドで作成されます。

```
DBG> SAVE CALLS AS CALLS4
```

新しいディスプレイはペーストボードから除去されます。その内容を表示するには DISPLAY コマンドを使用します。

```
DBG> DISPLAY CALLS4
```

EXTRACT コマンドには 2 つの用途があります。 1 つはディスプレイの内容をテキスト・ファイルに保存することです。たとえば，次のコマンドはディスプレイ CALLS の内容を抽出し，そのテキストをファイル COB34.TXT へ追加します。

```
DBG> EXTRACT/APPEND CALLS COB34
```

もう 1 つの用途は， EXTRACT/SCREEN_LAYOUT コマンドを使用してコマンド・プロシージャを作成し，それをあとのデバッグ・セッションで実行して前の画面状態を再作成することです。次の例では， EXTRACT/SCREEN_LAYOUT コマンドで省略時の指定 SYS$DISK:[ ]DBGSCREEN.COM を持つコマンド・プロシージャを作成します。このファイルには，現在の画面の状態を再作成するために必要なすべてのコマンドが含まれています。

```
DBG> EXTRACT/SCREEN_LAYOUT
   .
   .
   .
DBG> @DBGSCREEN
```

PROMPT ディスプレイは，他のディスプレイのように保存したりファイルの中へ抽出したりすることはできないので注意してください。

7.10 画面の高さと幅の変更

デバッグ・セッションの間，端末画面の高さまたは幅を変更することができます。 80 桁を表示した場合には折り返されてしまうような長い行を画面に収めたい場合や，ワークステーションでデバッガ・ウィンドウを他のウィンドウとの関係で再編集したい場合などです。

画面の高さまたは幅を変更するには， SET TERMINAL コマンドを使用します。このコマンドの一般的な効果は， VT シリーズの端末の場合でもワークステーションの場合でも同じです。

この例では，画面サイズが 80 桁 24 行の省略時の VT100 画面エミュレーション・モードでワークステーション・ウィンドウを使用して，デバッガをすでに起動し，画面モードで使用しているとします。この時点でより大きな画面を使用したい場合，次のコマンドを使用すれば画面の高さとデバッガ・ウィンドウの幅をそれぞれ 35 行と 110 桁に増やすことができます。

```
DBG> SET TERMINAL/PAGE:35/WIDTH:110
```

省略時の設定では，すべてのディスプレイが動的になっています。動的なディスプレイでは， SET TERMINAL コマンドで画面の高さまたは幅を変更すると，ウィンドウの寸法が比例して調整されます。したがって， SET TERMINAL コマンドを使用した場合は各ディスプレイの相対的な位置が保持されます。 DISPLAY コマンドで/[NO]DYNAMIC 修飾子を使用すると，ディスプレイを動的にするかしないかを制御できます。動的でないディスプレイの場合， SET TERMINAL コマンドの入力後にウィンドウ座標が変更されません。その後，ディスプレイを移動したりサイズ変更したりするために DISPLAY, MOVE, EXPAND の各コマンドや各種のキーパッド・キーの組み合わせが使用できます。

デバッガで使用している現在の端末の幅と高さを表示するには， SHOW TERMINAL コマンドを使用します。

デバッガの SET TERMINAL コマンドは， DCL レベルの端末画面サイズには影響を及ぼさないので注意してください。デバッガを終了した時点で，元の画面サイズに戻ります。

7.11 画面に関連した組み込みシンボル

次の組み込みシンボルは，言語式内でディスプレイと画面パラメータを指定する際に使用できます。

- %SOURCE_SCOPE— ソース・コードを表示する。%SOURCE_SCOPE の説明は第 7.4.1 項を参照。
- %INST_SCOPE— 命令を表示する。%INST_SCOPE の説明は第 7.4.4 項を参照。
- %PAGE, %WIDTH— 現在の画面の高さと幅を指定する。
- %CURDISP, %CURSCROLL, %NEXTDISP, %NEXTINST, %NEXTOUTPUT, %NEXTSCROLL, %NEXTSOURCE— 表示リスト内の表示を指定する。

7.11.1 画面の高さと幅

組み込みシンボルの%PAGE と%WIDTH は，それぞれ端末画面の現在の高さと幅を戻します。これらのシンボルはウィンドウ指定など，さまざまな式の中で使用できます。たとえば，次のコマンドは画面の中央付近の領域を占める MIDDLE という名前のウィンドウを定義します。

```
DBG> SET WINDOW MIDDLE AT (%PAGE/4,%PAGE/2,%WIDTH/4,%WIDTH/2)
```

7.11.2 ディスプレイ組み込みシンボル

DISPLAY コマンドで特定のディスプレイを参照するたびにディスプレイ・リストが更新され，必要な場合は順序も変更されます。最後に参照したディスプレイは最後にペーストボード上にペーストされるので，ディスプレイ・リストの末尾に置かれます。ディスプレイ・リストは SHOW DISPLAY コマンドを入力すれば表示できます。

ディスプレイ組み込みシンボルを使用すると，ディスプレイをそれらのディスプレイ・リスト内の相対的な位置で指定できます。これらのシンボルは，次に示すとおり，明示的なディスプレイ名ではなくディスプレイ・リスト内の相対的な位置によってディスプレイを参照できます。これらのシンボルは主にキーパッド・キー定義またはコマンド・プロシージャの中で使用します。

ディスプレイ・シンボルはディスプレイ・リストを循環リストとして扱います。したがって，ディスプレイ・シンボルを使用したコマンドを入力して，必要なディスプレイに到達するまでディスプレイ・リスト全体を循環することができます。

%CURDISP	現在のディスプレイ。これが DISPLAY コマンドで参照した最新のディスプレイで，最も隠されることの少ないディスプレイである。
%CURSCROLL	現在のスクロール・ディスプレイ。これが SCROLL, MOVE，および EXPAN D の各コマンドとそれに対応するキーパッド・キー (KP2, KP4, KP6，および KP8) 用の省略時のディスプレイである。
%NEXTDISP	リスト内で現在のディスプレイの後ろにある次のディスプレイ。次のディスプレイとは最上部のディスプレイに次ぐディスプレイである。ディスプレイ・リストは循環しているので，これはペーストボードの最下部にあるディスプレイで最も隠されることの多いディスプレイである。
%NEXTINST	ディスプレイ・リスト内で現在の機械語命令ディスプレイの後ろにある次の機械語命令ディスプレイ。現在の機械語命令ディスプレイとは EXAMINE/INSTRUCTION コマンドからの出力を受け取るディスプレイである。
%NEXTOUTPUT	ディスプレイ・リスト内で現在の出力ディスプレイの後ろにある次の出力ディスプレイ。出力ディスプレイは，まだ他のディスプレイへ出力されていないデバッガ出力を受け取る。
%NEXTSCROLL	ディスプレイ・リスト内で現在のスクロール・ディスプレイの後ろにある次のディスプレイ。
%NEXTSOURCE	ディスプレイ・リスト内で現在のソース・ディスプレイの後ろにある次のソース・ディスプレイ。現在のソース・ディスプレイは TYPE コマンドおよび EXAMINE/SOURCE コマンドからの出力を受け取るディスプレイである。

7.12 画面の寸法と定義済みウィンドウ

VT シリーズの端末では，画面は 24 行 80 桁または 132 桁で構成されます。ワークステーションでは，画面は高さにおいても幅においてもそれより大きくなります。デバッガは 100 行 255 桁までの画面サイズを収容できます。

デバッガには多くの定義済みウィンドウがあり，画面上でディスプレイの位置を設定するために使用できます。画面の完全な高さと幅の他に，定義済みウィンドウには，次の操作から求められる可能なすべての領域が含まれます。

- 縦方向のスクリーンの均等分割: 2 分割， 3 分割， 4 分割， 6 分割， 8 分割のいずれか
- 縦方向に連続した均等分割の結合: 2 分割， 3 分割， 4 分割， 6 分割， 8 分割のいずれか
- 左半分と右半分に分ける縦方向の分割

SHOW WINDOW コマンドは定義済みのすべてのディスプレイ・ウィンドウを識別します。

定義済みウィンドウの名前には次の規則が適用されます。接頭辞の L と R は，それぞれ左ウィンドウと右ウィンドウを表します。他の英字は全画面 (FS) または画面の高さ (H：半分， T：3 分の 1， Q：4 分の 1， S：6 分の 1， E：8 分の 1) を表しま

す。末尾の数字は画面の高さの場所 (最上部から始まる) を表します。次に例を示します。

- ウィンドウ T1, T2, T3 は，それぞれ画面の最上部，中央，最下部の 3 分の 1 を占める。
- ウィンドウ RH2 は，画面の右下半分を占める。
- ウィンドウ LQ23 は，画面の左中央の 2 つの 4 分の 1 を占める。
- ウィンドウ S45 は，画面の 4 番目と 5 番目の 6 分の 1 を占める。

次の 4 つのコマンドは，サイズと位置が同一 (スクリーンの上半分) であるウィンドウを持つディスプレイを作成します。

```
DBG> DISPLAY XYZ AT H1 SOURCE
DBG> DISPLAY XYZ AT Q12 SOURCE
DBG> DISPLAY XYZ AT S123 SOURCE
DBG> DISPLAY XYZ AT E1234 SOURCE
```

省略時の端末画面幅 (80 桁) の場合，定義済みウィンドウの左右の境界 (start-column, column-count) は次のとおりです。

- 左ウィンドウ: (1,40)
- 右ウィンドウ: (42,39)

表 7–3 は， 24 行という省略時の端末画面の高さに対して定義されている単一セグメント・ディスプレイ・ウィンドウの縦方向の境界 (start-line， line-count) を示しています。表 7–3 には， E23 (ディスプレイ・ウィンドウ E2 と E3 の組み合わせから作成されたディスプレイ・ウィンドウ) などのように，複数のセグメントで構成されるウィンドウは示されていません。

表 7–3 定義済みウィンドウ

ウィンドウ名	start-line,line-count	ウィンドウ位置
FS	(1,23)	全画面
H1	(1,11)	上半分
H2	(13,11)	下半分
T1	(1,7)	最上部の 3 分の 1
T2	(9,7)	中央の 3 分の 1
T3	(17,7)	最下部の 3 分の 1
Q1	(1,5)	最上部の 4 分の 1
Q2	(7,5)	2 番目の 4 分の 1
Q3	(13,5)	3 番目の 4 分の 1
Q4	(19,5)	最下部の 4 分の 1

(次ページに続く)

表 7–3 (続き) 定義済みウィンドウ

ウィンドウ名	start-line,line-count	ウィンドウ位置
S1	(1,3)	最上部の 6 分の 1
S2	(5,3)	2 番目の 6 分の 1
S3	(9,3)	3 番目の 6 分の 1
S4	(13,3)	4 番目の 6 分の 1
S5	(17,3)	5 番目の 6 分の 1
S6	(21,3)	最下部の 6 分の 1
E1	(1,2)	最上部の 8 分の 1
E2	(4,2)	2 番目の 8 分の 1
E3	(7,2)	3 番目の 8 分の 1
E4	(10,2)	4 番目の 8 分の 1
E5	(13,2)	5 番目の 8 分の 1
E6	(16,2)	6 番目の 8 分の 1
E7	(19,2)	7 番目の 8 分の 1
E8	(22,2)	最下部の 8 分の 1

7.13 各国に対応した画面モード

次のような論理名を定義することにより，画面モードに各国単位の機能を搭載することができます。

- DBG$SMGSHR — 画面管理 (SMG) の共有イメージを指定するときに使用。デバッガでは，画面モードを実現するときに SMG 共有イメージが使用される。アジア用の SMG 共有イメージは，マルチバイト文字を処理する。そのためデバッガでアジア用の SMG 共有イメージを使用すると，デバッガの画面モード・インタフェースで，マルチバイト文字の表示と処理が可能になる。

  DBG$SMGSHR 論理名を次のように定義する。

  ```
  $  DEFINE/JOB DBG$SMGSHR <name_of_Asian_SMG>
  ```

  <name_of_Asian_SMG>は，アジア用 OpenVMS の種類により異なる。たとえば，アジア用 SMG が日本版 OpenVMS の場合，名前は JSY$SMGSHR.EXE になる。

- SMG$DEFAULT_CHARACTER_SET — アジア用の SMG およびマルチバイト文字に使用。この論理名は， DBG$SMGSHR が定義された場合のみ定義する。この論理名を定義する方法については，アジア版または日本版画面管理ルーチンのマニュアルを参照。

第3部
DECwindowsインタフェース

第 3 部では，デバッガの DECwindows Motif for OpenVMS ユーザ・インタフェースについて説明します。

デバッガのコマンド・インタフェースについては，第 2 部をご覧ください。

8

DECwindows Motif インタフェースの概要

本章ではデバッガの DECwindows Motif for OpenVMS ユーザ・インタフェースについて説明します。コマンド・インタフェースについては，第 2 部を参照してください。

注意

OpenVMS デバッガ・バージョン 7.1 以上に対する DECwindows Motif for OpenVMS ユーザ・インタフェースでは， DECwindows Motif for OpenVMS バージョン 1.2 以上が必要です。

本章には次の内容が含まれます。

- ユーザ・インタフェースの各オプション (DECwindows Motif for OpenVMS インタフェースとコマンド・インタフェース) を含む OpenVMS デバッガの機能概要 (第 8.1 節)
- ウィンドウやメニューなどのデバッガの DECwindows Motif for OpenVMS 画面機能 (第 8.2 節)
- コマンド入力プロンプトでのデバッガ・コマンドの入力方法 (第 8.3 節)
- オンライン・ヘルプのアクセス方法 (第 8.4 節)

デバッグ・セッションの開始方法についての詳しい説明は，第 9 章を参照してください。デバッガの使用方法について説明は，第 10 章を参照してください。本章のプログラムのソース・コード EIGHTQUEENS.EXE については，付録 D を参照してください。

8.1 はじめに

OpenVMS デバッガでは，ワークステーション用に DECwindows Motif for OpenVMS グラフィカル・ユーザ・インタフェース (GUI) を使用できます。このインタフェースは画面モード・コマンド・インタフェースの拡張機能であり，マウスを使用してメニューから項目を選択したり，プッシュ・ボタンを有効または無効に設定したり，ポインタをドラッグしてウィンドウ内のテキストを選択できます。デバッガの DECwindows Motif for OpenVMS GUI メニューとプッシュ・ボタンを使用すると，基本的なほとんどのデバッグ作業を実行できます。

DECwindows Motif for OpenVMS GUI はキャラクタ・セル・コマンド・インタフェースの上に構築された GUI であり，**コマンド行**にコマンド入力プロンプトが表示されます (**コマンド・ビュー**)。 DECwindows Motif for OpenVMS GUI コマンド行から，次の目的でデバッガ・コマンドを入力できます。

- DECwindows Motif for OpenVMS ユーザ・インタフェースのメニューとプッシュ・ボタンで使用できる一部の操作を実行できる。
- DECwindows Motif for OpenVMS GUI メニューとプッシュ・ボタンでは使用できないデバッグ作業を実行するために入力できる。

他のデバッグ・コマンドを新しいプッシュ・ボタンや既存のプッシュ・ボタンに割り当てるために， DECwindows Motif for OpenVMS GUI をカスタマイズできます。

DECwindows Motif for OpenVMS GUI はローカル・モードまたはクライアント/サーバ・モードで実行することができます。クライアント/サーバ・モードでは，他の OpenVMS ノードからプログラムをリモートでデバッグできます。どちらの Motif モードのユーザ・インタフェースも，実質的に同じものです。インタフェースの起動方法については第 9 章で解説しています。

注意

DECwindows Motif for OpenVMS では，コマンド入力プロンプトに対する入力として HELP コマンドを認識しません。デバッガ・コマンドに関するオンライン・ヘルプが必要なときは，「Help」メニューで「On Commands」を選択してください。

コマンド行インタプリタ (CLI) なしで実行されるプリント・シンビオントなど，独立プロセスをデバッグするために DECwindows Motif for OpenVMS GUI を使用することはできません。 CLI のない独立プロセスのデバッグについては，第 1.11 節を参照してください。

8.1.1 便利な機能

デバッガの省略時の DECwindows Motif for OpenVMS インタフェースに用意されているいくつかの便利な機能について説明します。第 8.2 節では，図を使用して説明します。(デバッガのコマンド・インタフェースの便利な機能については，第 1.1.2 項を参照してください。)

◆ソース・コード・ディスプレイ

OpenVMS デバッガはソース・レベルのデバッガです。ソース・レベルのデバッガです。デバッガの**ソース・ビュー**には，プログラムの実行が一時停止しているとき，その命令の前後のソース・コードが表示されます。コンパイラ生成行番号を表示するかしないかはユーザが選択できます。

ソース・ブラウザには，次の機能があります。

- プログラムのイメージ，モジュール，ルーチンのリスト
- 選択したモジュールまたはルーチンのソース・コードの表示
- 下位の階層にあるモジュールやルーチンの表示
- 選択したルーチンをダブルクリックすることによるブレークポイントの設定

◆呼び出しスタック間のナビゲーション
メイン・ウィンドウの「Call Stack」メニューには，現在呼び出しスタックに入れられているルーチン呼び出しが表示されます。「Call Stack」メニューでルーチン名をクリックすることにより，(そのルーチンに対して) 次の事項のコンテキスト (有効範囲) を設定します。

- (ソース・ビューの) ソース・コード・ディスプレイ
- (**レジスタ・ビュー**の) レジスタ・ディスプレイ
- (**命令ビュー**の) 命令ディスプレイ
- シンボル検索

◆ブレークポイント
ブレークポイントを設定したり，オン・オフを切り替えたりするには，ソース・ビューまたは命令ビューの，ソース行の隣にあるボタンをクリックして選択します。ウィンドウのプルダウン・メニュー，ポップアップ・メニュー，コンテキスト依存のメニュー，ダイアログ・ボックスから項目を選択してブレークポイントを設定したり，オン・オフを切り替えたりすることもできます。また，条件付きブレークポイントを設定して，指定した条件が真になったときにプログラムの実行が停止するようにしたり，アクション・ブレークポイントを設定して，ブレークポイントでプログラムが停止するときに，1つまたは複数のデバッガ・コマンドを実行するようにしたりすることもできます。メイン・ウィンドウのボタン，命令ビューのプッシュ・ボタン，および**ブレークポイント・ビュー**を見ると，ブレークポイントのオン・オフや条件付きブレークポイントが識別できるようになっています。

◆プッシュ・ボタン
プッシュ・ボタン・ビューのプッシュ・ボタンは一般的な操作を制御します。プッシュ・ボタンをクリックすることにより，実行の開始，次のソース行のステップ実行，ウィンドウ内で選択した変数の値の表示，実行の割り込みなどが行えます。

ユーザは，プッシュ・ボタンおよびそれに対応するデバッガ・コマンドについて，変更，追加，削除，並べ替えなどができるようになっています。

◆コンテキスト依存のポップアップ・メニュー
コンテキスト依存のポップアップ・メニューには，(ソース・ビュー，コマンド・ビューなどの) それぞれのビューに対応した一般的な操作が表示されます。MB3 をクリックすると，ポップアップ・メニューが現れ，選択しているテキスト，ポイントしているソース行，作業中のビューなどに対するアクションがリストされます。

◆データの表示と操作

変数または式の値を表示するときは，ソース・ビューで変数や式を選択してから，「Examine」(変数のテスト) などのプッシュ・ボタンをクリックします。ウィンドウのプルダウン・メニュー (たとえば「Commands」プルダウン・メニューの「Examine」)，コンテキスト依存のメニュー，ダイアログ・ボックスなどから項目を選択することにより，選択した値を表示することもできます。値はいろいろな型や基数で表示できます。

変数の値を変更するには，**モニタ・ビュー**に現在表示されている値を編集します。ウィンドウのプルダウン・メニュー (たとえば「Commands」プルダウン・メニューの「Deposit」)，コンテキスト依存のメニュー，ダイアログ・ボックスなどから項目を選択して値を変更することもできます。

プログラムからデバッガに制御が戻ると，指定されている変数の更新済みの値がモニタ・ビューに表示されます。

◆保持デバッガの RERUN コマンド

保持デバッガ (kept debugger) でデバッガを動作させることにより，デバッガを終了せずに，同じプログラムを再実行したり，別のプログラムを実行したりすることができます。プログラムを再実行する場合，ブレークポイント，トレースポイント，静的ウォッチポイントの現在の状態を保存することができます。保持デバッガは，画面モード・デバッガでも使用できます。保持デバッガの詳細については，第 9.1 節を参照してください。

◆クライアント/サーバ構成

デバッガをクライアント/サーバ構成で実行すると， OpenVMS ノード上で実行されているプログラムを， DECwindows Motif for OpenVMS インタフェースを使用する別の OpenVMS ノードまたは Microsoft Windows インタフェースを使用する PC からリモートでデバッグすることができます。同じデバッグ・サーバに 31 個までのデバッグ・クライアントが同時にアクセスできるため，多様なデバッグ・オプションを利用できます。

◆命令ビューとレジスタ・ビュー

命令ビューには，プログラムのデコード済み命令ストリーム (実際に実行されているコード) が表示されます。デバッグ対象のプログラムがコンパイラによって最適化されているため，ソース・ビューのソース・コードが実行中のコードに反映されていない場合などに，このビューを使用すると便利です。命令にブレークポイントを設定したり，各命令に対応しているメモリ・アドレスやソース・コード行番号を表示したりできます。

レジスタ・ビューには，すべての機械語レジスタの現在の内容が表示されます。レジスタに別の値を格納するために，表示されている値を書き換えることができます。

◆デバッガ・ステータス・インジケータ
デバッガには，デバッガの状態を示すステータス・インジケータがあります。このインジケータは，次のいずれかの状態を示します。

- D— デバッグ対象のプログラムを実行中。
- U— デバッガはユーザ・コマンドを実行中。

◆スレッド・プログラムのサポート
スレッド・ビューには，マルチスレッド・プログラムのすべてのタスクの現在の状態が表示されます。スレッドの実行，優先順位，状態変化などを制御するために，スレッド特性を変更することができます。

◆コマンド・インタフェースとの統合
デバッガの DECwindows Motif for OpenVMS インタフェースは，画面モード・デバッガを強化したものです。これは，コマンド方式の画面モード・デバッガの上に設けられており，この 2 つは密接に統合されています。

- DECwindows Motif for OpenVMS GUI メニューおよびプッシュ・ボタンを使用した場合，コマンドがコマンド・ビューにエコーバックされ，アクションの記録が可能になる。
- プロンプトにコマンドを入力すると，それらのコマンドに対応して DECwindows Motif for OpenVMS のビューが更新される。

◆ソース・レベルのエディタとの統合
デバッガを終了せずに，プログラムのソース・コードを編集することができます。エディタ・ビューでは，ソース・コードの表示の他，テキストの検索，置換，追加も可能になっています。エディタ・ビューのテキスト・バッファでは，新しいファイルや既存のファイルの間をすばやく移動したり，バッファ間でテキストをコピー，カット，ペーストすることができます。

デバッガの DECwindows Motif for OpenVMS メニュー・インタフェースで使用されるテキスト・エディタは，簡単な機能のエディタであるため，ランゲージ・センシティブ・エディタ (LSE) のような優れた機能を持つエディタにはおよびません。内蔵エディタ以外のエディタを使用する場合は，コマンド・ビューの DBG>プロンプトで Edit コマンドを入力します (EDIT コマンドを参照)。

◆カスタマイズ
ユーザはデバッガの DECwindows Motif for OpenVMS インタフェースの形態を変更することができます。また，デバッガ起動時の環境をカスタマイズするために，現在の設定を**リソース・ファイル**に保存することができます。変更できる要素の例を次に示します。

- ウィンドウとビューの構成 (たとえば，サイズ，画面の位置，順序)
- プッシュ・ボタンの順序，ラベル，それに対応するデバッガ・コマンド (プッシュ・ボタンやコマンドの追加および削除を含む)
- 表示されるテキストの文字フォント

◆オンライン・ヘルプ
デバッガの DECwindows Motif for OpenVMS インタフェースおよびコマンド・インタフェースでは，オンライン・ヘルプを利用することができます。このオンライン・ヘルプはコンテキスト依存のヘルプです。

8.2 デバッガのウィンドウとメニュー

次の各項では，デバッガのウィンドウ，メニュー，ビューの他， OpenVMS デバッガ DECwindows Motif for OpenVMS インタフェースのその他の機能について説明します。

8.2.1 省略時のウィンドウ構成

省略時の設定ではデバッガを起動すると，**メイン・ウィンドウ**に図 8–1 のように表示されます。

第 9.1 節で説明している方法でデバッガを起動したときの初期画面では，ソース・ビューは空になっています。プログラムをデバッガの制御下に置いた後のソース・ビューが図 8–1 に示されています。デバッガで特定のイメージ (この例では，EIGHTQUEENS) を実行すると，プログラムはデバッガの制御下に入ります。

起動時の構成をカスタマイズすることができます。詳しい説明は第 10.10.1 項を参照してください。

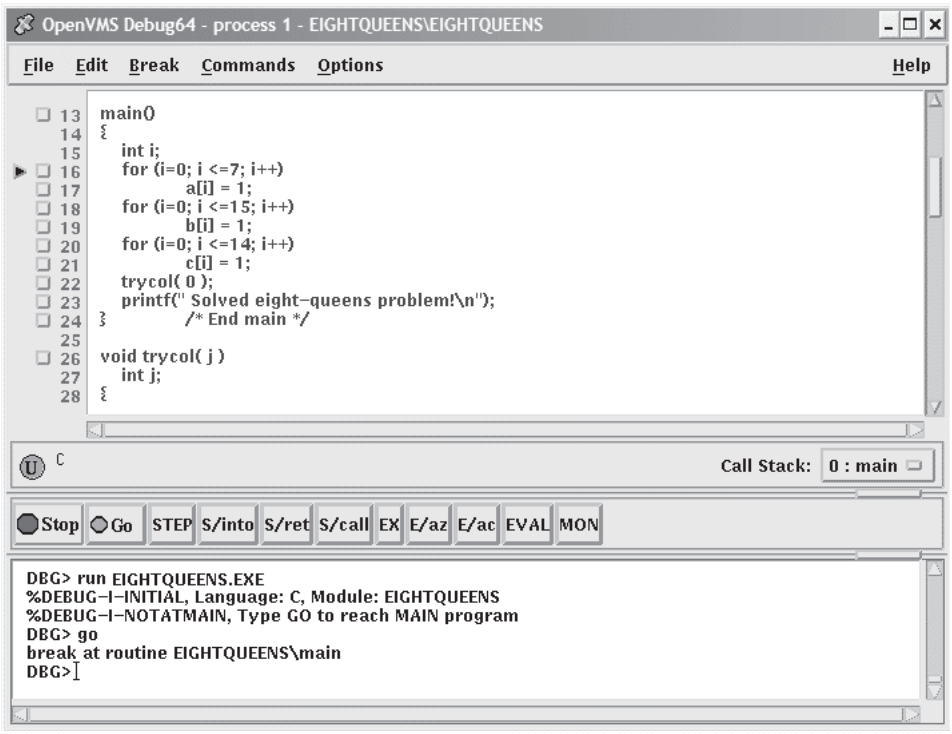
8.2.2 メイン・ウィンドウ

メイン・ウィンドウには次のものが含まれます (図 8–1 を参照してください)。

- タイトル・バー (第 8.2.2.1 項を参照)
- ソース・ビュー (第 8.2.2.2 項を参照)
- 呼び出しスタック・ビュー (第 8.2.2.4 項を参照)
- プッシュ・ボタン・ビュー (第 8.2.2.5 項を参照)
- コマンド・ビュー (第 8.2.2.6 項を参照)

デバッガが Alpha プロセッサまたは Integrity プロセッサ上で実行されている場合，デバッガの名前は「OpenVMS Debug64」になります。

図 8–1 デバッガのメイン・ウィンドウ



8.2.2.1 タイトル・バー

タイトル・バーは，メイン・ウィンドウの上部にあり，デバッガの名前 (省略時の場合)，デバッグするプログラムの名前，現在ソース・ビューに表示されているソース・コード・モジュールの名前などが表示されます。

8.2.2.2 ソース・ビュー

ソース・ビューには次のものが表示されます。

- デバッグ中のプログラムのソース・コード。省略時の設定では，(ソース・コードの左に) コンパイラ生成行番号が表示される。行番号を表示しないようにする方法については，第 10.1 節を参照。
- ブレークポイント・トグル用のプッシュ・ボタン。
- 現在のロケーションを示すポインタ (ブレークポイント・プッシュ・ボタンの左に表示される三角形)。これにより，プログラムの実行が再開するときに，実行されるソース・コードの行が示される。

ソース・コードの表示についての詳しい説明は，第 8.2.2.3 項および第 10.1 節を参照してください。

8.2.2.3 メイン・ウィンドウ上のメニュー

図 8–2 と表 8–1 では，メイン・ウィンドウ上のメニューについて説明します。

図 8–2 メイン・ウィンドウ上のメニュー

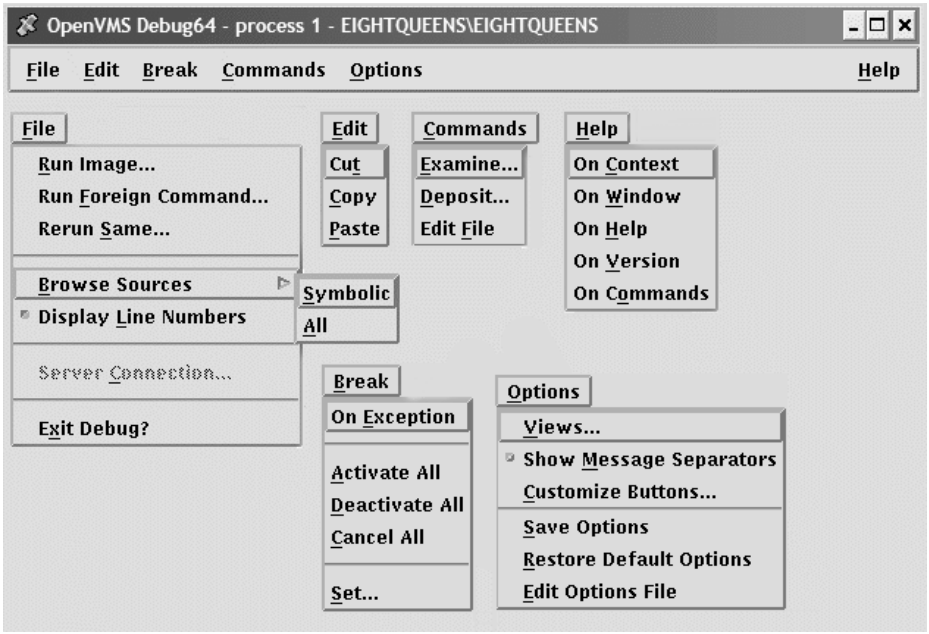


表 8–1 メイン・ウィンドウ上のメニュー

メニュー	項目	説明
File	Run Image...	実行可能なイメージを指定して，プログラムをデバッガの制御下に置く。
	Run Foreign Command...	フォーリン・コマンドのシンボルを指定して，プログラムをデバッガの制御下に置く。
	Rerun Same...	同じプログラムをデバッガの制御下で再実行する。
	Browse Sources	ユーザ・プログラムのモジュールのソース・コードを表示する。ルーチンにブレークポイントを設定する。 • Symbolic— デバッガがシンボリック情報を保有しているモジュールのみをリストに表示する。 • All— すべてのモジュールをリストに表示する。
	Display Line Numbers	ソース・ビューに行番号を表示する，または表示しないようにする。
	Server Connection...	(クライアント/サーバ・モード) 接続するサーバのネットワーク・バインディング文字列を指定する。

(次ページに続く)

表 8–1 (続き) メイン・ウィンドウ上のメニュー

メニュー	項目	説明
	Exit Debug?	デバッグ・セッションを終了し，デバッガを終了する。
Edit	Cut	選択されたテキストをカットし，クリップボードへコピーする。テキストは，フィールドまたは入力を受け付けるリージョンからのみカットできる (ただしほとんどの場合に， Cut を実行すると選択したテキストがクリップボードにコピーされる)。
	Copy	選択されたテキストをウィンドウからクリップボードへコピーするが，テキストの削除は行わない。
	Paste	テキスト入力フィールドまたはテキスト入力領域へ，クリップボードからテキストをペーストする。
Break	On Exception	プログラム実行中にシグナル通知された例外でブレークする。
	Activate All	以前に設定したブレークポイントをすべて有効にする。
	Deactivate All	以前に設定したブレークポイントをすべて無効にする。
	Cancel All	デバッガのブレークポイント・リストとブレークポイント・ビューからブレークポイントをすべて削除する。
	Set...	(特定の条件やアクションに関連付けられた) 新しいブレークポイントを指定の箇所に設定する。
Commands	Examine...	変数や式の現在値を調べる。出力値の型キャストまたは基数の変更ができる。
	Deposit...	変数に値を格納する。入力値の基数を変更できる。
	Edit File	デバッガのエディタでユーザ・ファイルのソース・コードを編集する。
Options	Views...	次のビューの 1 つまたはいくつかを表示する。 ブレークポイント・ビュー モニタ・ビュー レジスタ・ビュー タスキング・ビュー 命令ビュー
	Track Language Changes	デバッガが前回実行されたモジュールとは異なる言語で書かれたモジュールに入ったときに通知を行う。
	Show Message Separators	デバッガが表示する個々のコマンドとメッセージの間に点線を表示する。
	Customize Buttons...	プッシュ・ボタン・ビューのプッシュ・ボタンやそれに対応したデバッガ・コマンドの変更，追加，削除，並べ替えなどを行う。
	Save Options	ウィンドウとビューの構成やプッシュ・ボタンの定義など，会話形式でカスタマイズ可能な DECwindows Motif for OpenVMS 機能の現在の設定を保存する。これにより，次にデバッガを起動するとき，デバッガの現在の構成が維持されるようになる。

(次ページに続く)

表 8–1 (続き) メイン・ウィンドウ上のメニュー

メニュー	項目	説明
	Restore Default Options	システムの省略時デバッガ・リソース・ファイル DECW$SYSTEM_DEFAULTS:VMSDEBUG.DAT をユーザ指定リソース・ファイル DECW$USER_DEFAULTS:VMSDEBUG.DAT にコピーする。省略時のオプションは，次にデバッガを起動するときに有効になる。
	Edit Options File	デバッグ・エディタで，ユーザ指定リソース・ファイル DECW$USER_DEFAULTS:VMSDEBUG.DAT のロードと表示を行い，表示と変更ができるようにする。
Help	On Context	コンテキスト依存のオンライン・ヘルプを表示可能にする。
	On Window	デバッガについての情報を表示する。
	On Help	オンライン・ヘルプ・システムについての情報を表示する。
	On Version	デバッガのこのバージョンについての情報を表示する。
	On Commands	デバッガ・コマンドについての情報を表示する。

表 8–2 レジスタ・ビューでの表示

レジスタ・タイプ	Alpha 表示	Integrity 表示
Call Frame	R0, R25, R26, R27, FP, SP, F0, F1, PC, PS, FPCR, SFPCR	PC, CFM, BSP, BSPSTORE, PFS, RP, UNAT, GP, SP, TP, AI
General Purpose	R0-R28, FP, SP, R31	PC, GP, R2-R11, SP, TP, R14-R24, AI, R26-R127
Floating Point	F0-F31	F2 - F127

8.2.2.4 「Call Stack」メニュー

「Call Stack」メニューは，ソース・ビューとプッシュ・ボタン・ビューの間にありますが，ここには，ソース・ビューに表示されているソース・コードのルーチン名が表示されます。このメニューでは，現在スタックに入れられているルーチン呼び出しがリストされ，ソース・コード表示とシンボル検索の有効範囲をスタック上の任意のルーチンに設定できるようになっています (第 10.6.2 項を参照)。

8.2.2.5 プッシュ・ボタン・ビュー

図 8–3 および表 8–3 では，メイン・ウィンドウ内の省略時のプッシュ・ボタンについて説明します。第 10.10.3 項で説明しているように，ボタンおよびボタンに対応したデバッガ・コマンドを変更，追加，削除したり，並べ替えたりできます。

図 8–3 プッシュ・ボタン・ビューの省略時のボタン

表 8–3 プッシュ・ボタン枠の省略時のボタン

ボタン	説明
Stop	デバッグ・セッションを終了せずに，プログラムの実行やデバッガによる操作に割り込みをかける
Go	現在のプログラム記憶位置から，実行を開始または再開する。
STEP	プログラムをステップ単位で 1 ステップ実行する。省略時の設定では，ソース・コードの中の実行可能な 1 行が 1 ステップになる。
S/in	ルーチン呼び出し文で実行が一時停止された場合，呼び出されたルーチンの開始地点の直後へ実行を移す。ルーチン呼び出し文でなければ， STEP と同じ動作をする。
S/ret	現在のルーチンの終了地点へ直接プログラムの実行を移す。
S/call	次の Call 命令または Return 命令へ直接プログラムの実行を移す。
EX	ウィンドウ内でユーザが名前を選択した変数の，現在の値をコマンド・ビューに表示する。
E/az	ウィンドウ内でユーザが名前を選択した変数の，現在の値をコマンド・ビューに表示する。変数は， 0 で終了する ASCII 文字列に変換される。
E/ac	ウィンドウ内でユーザが名前を選択した変数の，現在の値をコマンド・ビューに表示する。変数は，長さを指定された ASCII 文字列に変換される。この文字列では，文字列の長さを示す 1 バイトのカウント・フィールドが前に付いている。
EVAL	現在の言語 (省略時の場合，メイン・プログラムを含むモジュールの言語) の式の値をコマンド・ビューに表示する。
MON	ウィンドウで選択した変数名とその変数の現在の値をモニタ・ビューに表示する。プログラムからデバッガに制御が戻ると，デバッガは値を自動的にチェックし，表示されている値をそれに応じて更新する。

8.2.2.6 コマンド・ビュー

メイン・ウィンドウのプッシュ・ボタンのすぐ下に位置するコマンド・ビューには，コマンド行にタイプしたコマンド入力が入れられ (第 8.3 節を参照)，オプション・ビューに表示される情報以外のデバッガ出力が表示されます。たとえば，次のような出力がメッセージ領域に表示されます。

- 操作の結果
- 診断メッセージ (デバッガの診断メッセージに関するオンライン・ヘルプについての説明は，第 8.4.4 項を参照)
- コマンドのエコーバック (デバッガはユーザの DECwindows Motif for OpenVMS のメニューとプッシュ・ボタンをデバッガ・コマンドに変換し，これらのコマンドをコマンド・ビューのコマンド行に表示し，最近使用したコマンドを記録する。こうすることにより，ユーザの入力とデバッガのアクションとを相互に関連づけることができる。)

ポップアップ・メニューから「Clear Command Window」を選択することにより，コマンド・ビュー全体をクリアして，現在のコマンド行プロンプトだけを残しておくことができます。

ポップアップ・メニューから「Clear Command Line」を選択することにより，現在のコマンド行をクリアすることができます。

8.2.3 オプション・ビュー・ウィンドウ

表 8–4 にオプション・ビューの一覧を示します。メイン・ウィンドウの「Options」メニューから「Views...」を選択するとオプション・ビューへアクセスできます。

表 8–4 オプション・ビュー

ビュー	説明
ブレークポイント・ビュー	現在設定されているすべてのブレークポイントの一覧を表示する。ブレークポイントが有効か無効か，または条件付きブレークポイントとして設定されているかどうかを示す。ブレークポイント・ビューでは，各ブレークポイントの状態を変更することもできる。
モニタ・ビュー	プログラムの実行時に値をモニタしたい変数の一覧を表示する。プログラムからデバッガに制御が戻ると (たとえば，1 ステップ実行したあと，またはブレークポイントに到達したとき)，デバッガは表示されている値を更新する。あるいは，ウォッチポイントを設定して，特定の変数が変更されるたびに実行を停止させることができる。また，ユーザが変数の値を変更することもできる。
命令ビュー	ユーザ・プログラムのデコード済み命令ストリームを表示する。ユーザは命令にブレークポイントを設定することができる。省略時の設定では，対応するメモリ・アドレスとソース行番号が命令の左側に表示される。このアドレスと行番号を表示しないように選択することもできる。
レジスタ・ビュー	すべての機械語レジスタの現在の内容を表示する。プログラムからデバッガに制御が戻ると，デバッガは表示されている値を更新する。レジスタ・ビューでは，ユーザがレジスタ内の値を変更することもできる。
タスキング・ビュー	タスキング・プログラムのすべての存在している (未終了) タスクの一覧を表示する。また，各タスクについての情報が表示されるので，ユーザは各タスクの状態を変更することができる。

図 8–5 では，「View」メニューを図 8–4 のように選択した結果として表示される，ブレークポイント・ビュー，モニタ・ビュー，レジスタ・ビューの構成の例を示します。

図 8–6 では，命令ビューを示しています。このウィンドウは，分割ウィンドウになっているため，使いやすい位置に配置することができます。図 8–7 は，タスキング・ビューを示しています。

表示されるレジスタと命令はシステム固有なので注意してください。図 8–5 と図 8–6 は， Integrity 固有のレジスタおよび命令です。

すべてのウィンドウは移動したりサイズを変更できます。デバッガを再起動したときにウィンドウとビューの特定の構成が自動的に設定されるように，その構成を保存しておくこともできます (第 10.10.1 項を参照)。

注意

UI アプリケーションをデバッグするときに，多くのデバッガ・ウィンドウがユーザ・プログラムのウィンドウと重なり合う場合には，X サーバはユーザ・プログラムを異常終了させることがあります。

この問題を回避するには，ユーザ・プログラムに属すウィンドウにデバッガのウィンドウが重ならないようにしてください。

図 8–4 デバッガ・メイン・ウィンドウ

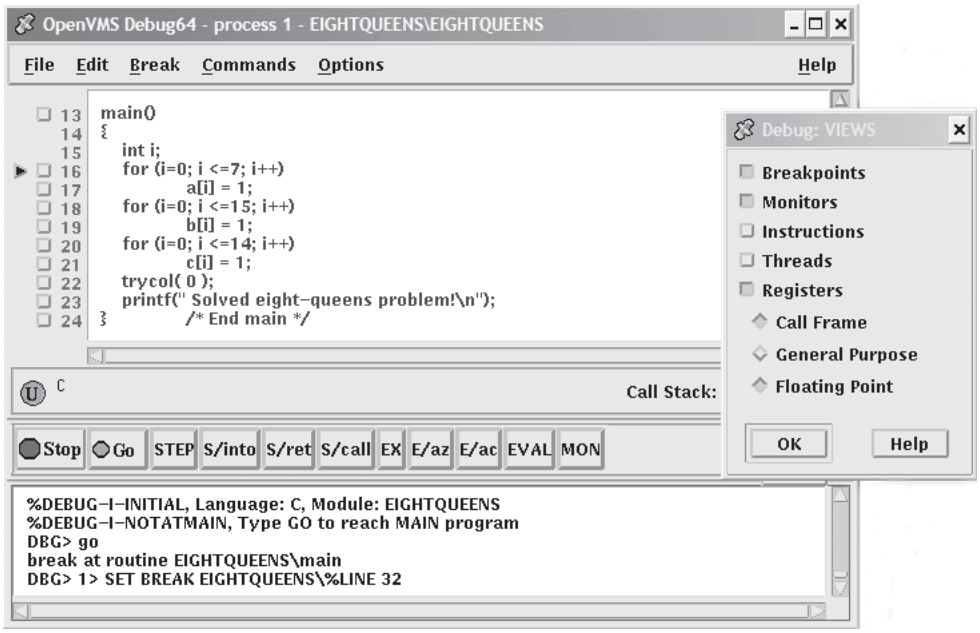
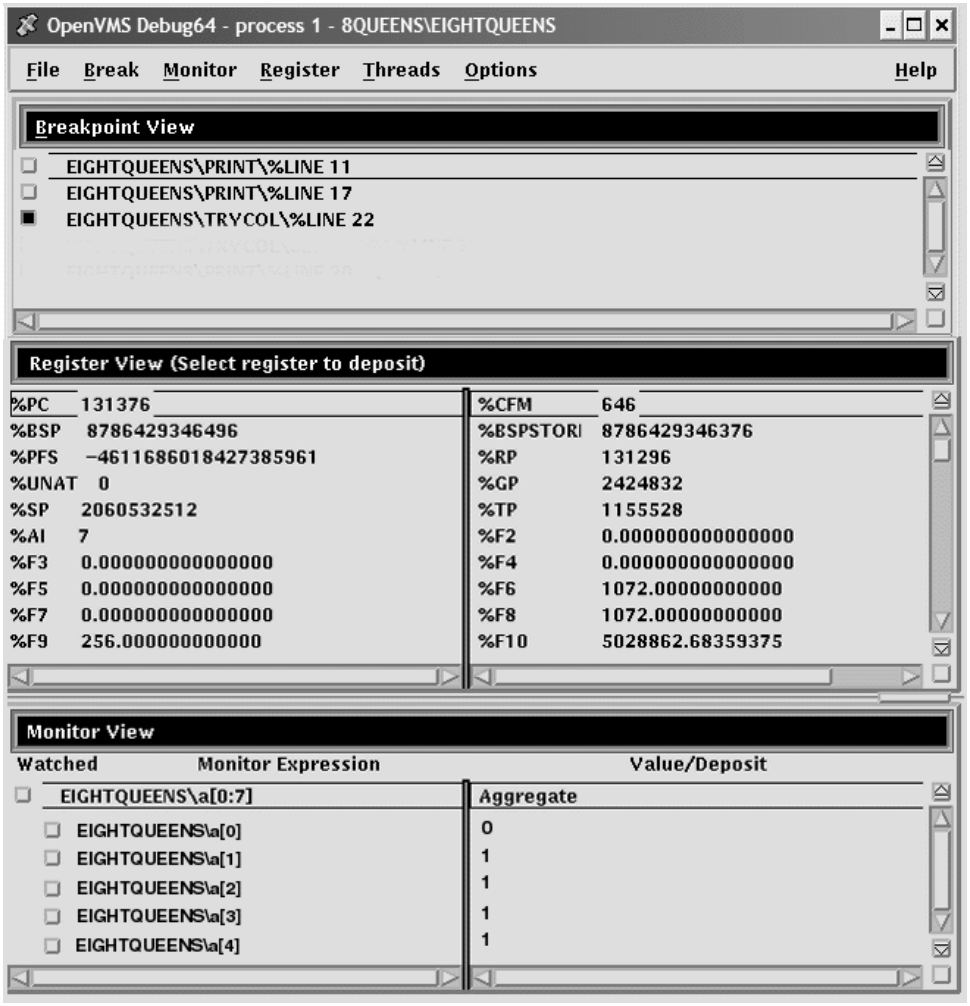


図 8–5 ブレークポイント・ビュー，モニタ・ビュー，レジスタ・ビュー



8.2.3.1 オプション・ビュー・ウィンドウのメニュー

図 8–8 と表 8–5 では，オプション・ビュー・ウィンドウ上のメニューについて説明します。

図 8–6 命令ビュー

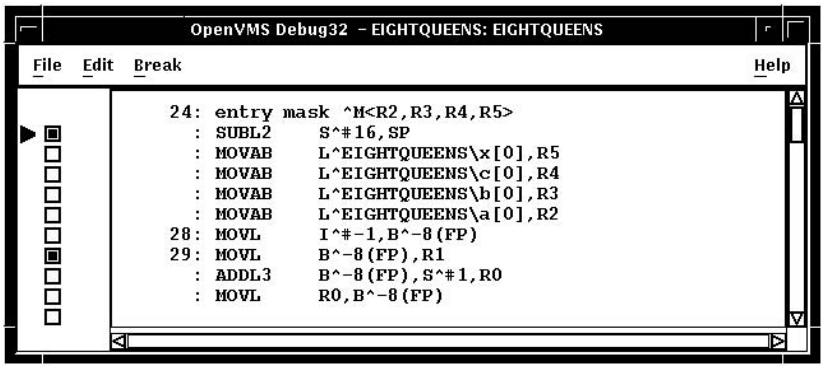


図 8–7 スレッド・ビュー

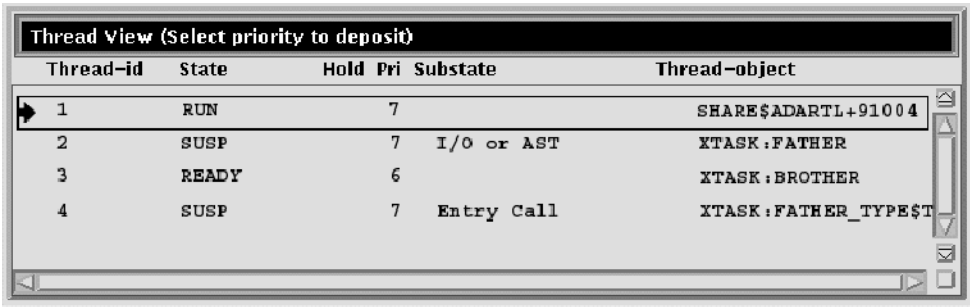


図 8–8 オプション・ビュー・ウィンドウ上のメニュー

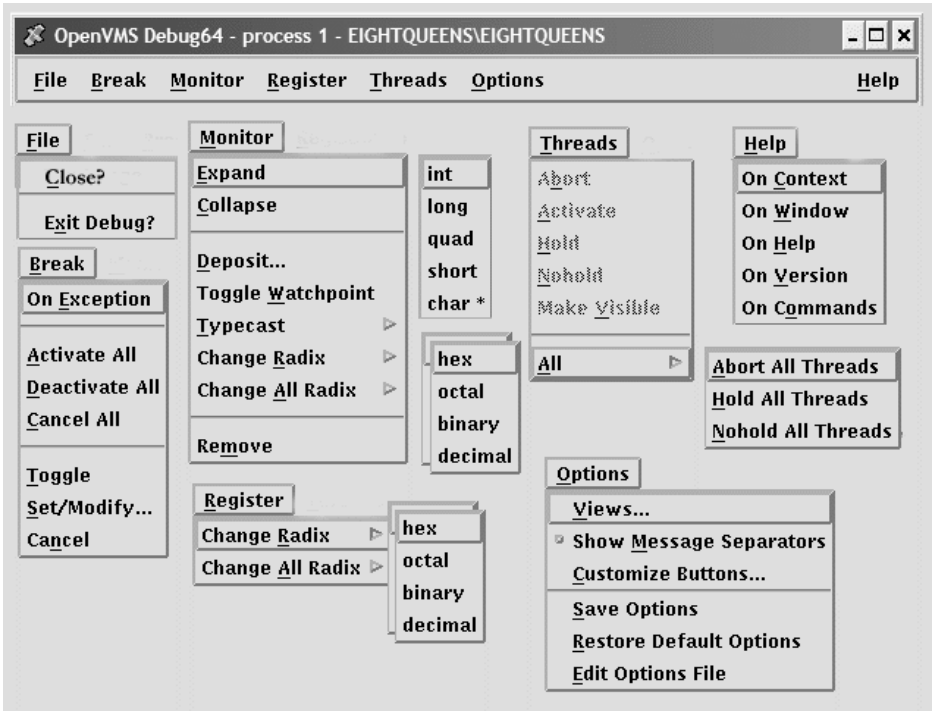


表 8–5 オプション・ビュー・ウィンドウ上のメニュー

メニュー	項目	説明
File	Close	オプション・ビュー・ウィンドウをクローズする。
	Exit Debugger	デバッグ・セッションを終了し，デバッガを終了する。
Break	On Exception	プログラム実行中にシグナル通知された例外でブレークする。
	Activate All	以前に設定したブレークポイントをすべて有効にする。
	Deactivate All	以前に設定したブレークポイントをすべて無効にする。
	Cancel All	デバッガのブレークポイント・リストとブレークポイント・ビューからブレークポイントをすべて削除する。
	Toggle	ブレークポイントの有効と無効を切り替える。
	Set/Modify...	(特定の条件やアクションに関連付けられた) 新しいブレークポイントを指定の箇所に設定する。
	Cancel	個々のブレークポイントをキャンセル (削除) する。
Monitor	Expand	モニタ・ビューの出力を拡大し，選択されている項目の各構成要素の値と集合体の値を表示する。
	Collapse	モニタ・ビューの出力を縮小し，選択されている項目の各構成要素の値は表示せず，集合体の値だけを表示する。
	Deposit...	モニタしている要素の値を変更する。
	Toggle Watchpoint	選択されたウォッチポイントの有効と無効を切り替える。
	Typecast	サブメニューを使用し，選択された変数の出力を int, long, quad, short, または char*に型キャストする。
	Change Radix	サブメニューを使用し，選択された変数の出力の基数を 16, 8, 10, または 2 に変更する。
	Change All Radix	サブメニューを使用し，以後モニタするすべての要素の出力の基数を 16, 8, 10, または 2 に変更する。
	Remove	モニタ・ビューから要素を削除する。
Register	Change Radix	サブメニューを使用し，選択されたレジスタの基数を 16, 8, 10, または 2 に変更する。
	Change All Radix	サブメニューを使用し，すべてのレジスタの基数を 16, 8, 10, または 2 に変更する。
Tasks	Abort	選択されたタスクを次の可能な機会に終了させる。
	Activate	選択されたタスクをアクティブ・タスクにする。
	Hold	選択されたタスクを保留にする。
	Nohold	選択されたタスクの保留を解除する。
	Make Visible	選択されたタスクを可視タスクにする。
	All	サブメニューを使用し，すべてのタスクを強制終了するか，またはすべてのタスクの保留を解除する。

(次ページに続く)

表 8–5 (続き) オプション・ビュー・ウィンドウ上のメニュー

メニュー	項目	説明
Options	Views...	次のビューの 1 つまたはいくつかを表示する。 ブレークポイント・ビュー モニタ・ビュー レジスタ・ビュー タスキング・ビュー 命令ビュー
	Customize Buttons...	プッシュ・ボタン・ビューのプッシュ・ボタンやそれに対応したデバッガ・コマンドの変更，追加，削除，並べ替えを行う。
	Save Options	ウィンドウとビューの構成やプッシュ・ボタンの定義など，会話形式でカスタマイズ可能な DECwindows Motif for OpenVMS 機能の現在の設定を保存する。これにより，次にデバッガを起動するとき，デバッガの現在の構成が維持されるようになる。
	Restore Default Options	システムの省略時デバッガ・リソース・ファイル DECW$SYSTEM_DEFAULTS:VMSDEBUG.DAT をユーザ指定リソース・ファイル DECW$USER_DEFAULTS:VMSDEBUG.DAT にコピーする。省略時のオプションは，次にデバッガを起動するときに有効になる。
	Edit Options File	デバッグ・エディタで，ユーザ指定リソース・ファイル DECW$USER_DEFAULTS:VMSDEBUG.DAT のロードと表示を行い，表示と変更ができるようにする。
Help	On Context	コンテキスト依存のオンライン・ヘルプを表示可能にする。
	On Window	OpenVMS デバッガについての情報を表示する。
	On Help	オンライン・ヘルプ・システムについての情報を表示する。
	On Version	デバッガのこのバージョンについての情報を表示する。
	On Commands	デバッガ・コマンドについての情報を表示する。

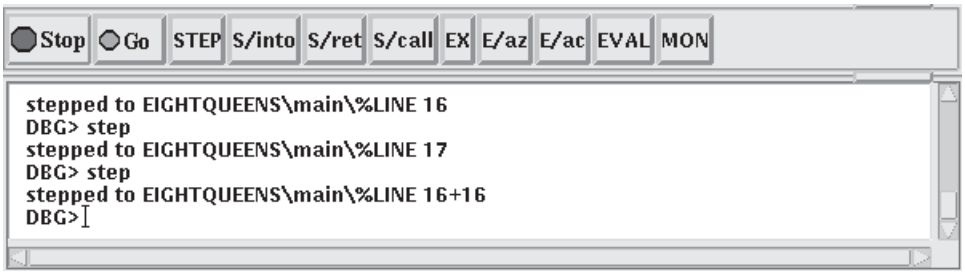
8.3 プロンプトでのコマンドの入力

DECwindows Motif for OpenVMS インタフェースはコマンド・インタフェースの上に設けられています。コマンド行は，コマンド・ビューの最終行にあり，コマンド入力プロンプト (DBG>) によって識別されますが，次の場合，このプロンプトにデバッガ・コマンドを入力することができます。

- ある種の操作において DECwindows Motif for OpenVMS インタフェースのメニューとプッシュ・ボタンを使用する代わりにコマンドを入力する場合
- DECwindows Motif for OpenVMS インタフェースのプルダウン・メニューとプッシュ・ボタンからは利用できないデバッグ・タスクを行う場合

図 8–9 に，コマンド・ビューで RUN コマンドを実行した状態を示します。

図 8–9 プロンプトでのコマンドの入力



DECwindows Motif for OpenVMS インタフェースのプルダウン・メニューとプッシュ・ボタンを使用する場合，デバッガはユーザの入力をデバッガ・コマンドに変換し，これらのコマンドをコマンド行にエコーバックします。その結果，ユーザがコマンドを記録できるようになります。エコーバックされたコマンドと，ユーザが明示的に入力したコマンドとを視覚的に区別することはできません。

デバッガのコマンド・インタフェースについての詳しい説明は，第 2 部を参照してください。各コマンドのオンライン・ヘルプについての説明は，第 8.4.3 項を参照してください。

デバッガ・コマンドはプロンプトで対話的に入力するだけでなく，デバッガ初期化ファイルとコマンド・ファイルに格納しておいて， DECwindows Motif for OpenVMS 環境内で実行することができます。

また，コマンド入力プロンプトで使用できるキーパッド・サポートを利用することもできます。このキーパッド・サポートは，コマンド・インタフェースに用意されている多様なキーパッド・サポートのサブセットです。キーパッド・サポートについては付録 A で説明します。コンピュータのキーパッドの個々のキーに割り当てられているコマンドを表 8–6 に示します。

表 8–6 DECwindows Motif for OpenVMS デバッガ・インタフェースのキーパッド定義

コマンド	対応するキー
Step/Line	KP0
Step/Into	GOLD-KP0
Step/Over	BLUE-KP0
Examine	KP1

(次ページに続く)

表 8–6 (続き) DECwindows Motif for OpenVMS デバッガ・インタフェースのキーパッド定義

コマンド	対応するキー
Examine^	GOLD-KP1
Go	KP,
Show Calls	KP5
Show Calls 3	GOLD-KP5

このいずれかのコマンドを入力するには，各コマンドに対応する，キーパッド上の1つまたは複数のキーを押してから ENTER キーを押します (GOLD キーは PF1, BLUE キーは PF4 です)。

キーのバインディングの変更，またはコマンドを割り当てられていないキーパッド・キーへの割り当てについては，第 10.10.4.4 項を参照してください。

8.3.1 DECwindows Motif for OpenVMS インタフェース内で使用不可能なデバッガ・コマンド

表 8–7 に，デバッガの DECwindows Motif for OpenVMS インタフェースで使用できないデバッガ・コマンドの一覧を示します。ここに示すコマンドのほとんどは，デバッガの画面モードと対応しているものです。

表 8–7 DECwindows Motif for OpenVMS ユーザ・インタフェースで使用不可能なデバッガ・コマンド

ATTACH	SELECT
CANCEL MODE	(SET,SHOW) ABORT_KEY
CANCEL WINDOW	(SET,SHOW) KEY
DEFINE/KEY	(SET,SHOW) MARGINS
DELETE/KEY	SET MODE [NO]KEYPAD
DISPLAY	SET MODE [NO]SCREEN
EXAMINE/SOURCE	SET MODE [NO]SCROLL
EXPAND	SET OUTPUT [NO]TERMINAL
EXTRACT	(SET,SHOW) TERMINAL
HELP[1]	(SET,SHOW) WINDOW
MOVE	(SET,CANCEL) DISPLAY
SAVE	SHOW SELECT
SCROLL	SPAWN

[1]各コマンドについてのヘルプは，デバッガのウィンドウの「Help」メニューから利用できます。

ユーザがこれらのコマンドをコマンド行に入力した場合や，デバッガがコマンド・プロシージャを実行しているときにこれらのコマンドが見つかった場合，デバッガにエラー・メッセージが表示されます。

8.4 デバッガについてのオンライン・ヘルプの表示

デバッグ・セッションの途中で，デバッガとデバッグについて次のオンライン・ヘルプを使用することができます。

- コンテキスト依存のヘルプ — ウィンドウまたはダイアログ・ボックス内の領域やオブジェクトについての情報。
- タスク用ヘルプ — 「Overview of the Debugger」という入門用のヘルプ・トピックと，特定のデバッグ・タスクについてのサブトピックで構成されている。
- デバッガ・コマンドやいろいろなトピック (言語サポートなど) についてのヘルプ。
- デバッガの診断メッセージについてのヘルプ。

コンテキスト依存のトピックに関連するタスク用トピックは，ヘルプ・ウィンドウ内の補助トピックの一覧によって結びつけられています。

8.4.1 コンテキスト依存のヘルプの表示

コンテキスト依存のヘルプは，ウィンドウやダイアログ・ボックス内の領域またはオブジェクトについての情報です。

コンテキスト依存のヘルプを表示するには，次の手順に従ってください。

1. デバッガのウィンドウの「Help」メニューから「Context」を選択する。ポインタの形が疑問符 (?) に変わる。
2. デバッガのウィンドウまたはダイアログ・ボックス内のオブジェクトまたは領域に疑問符を置く。
3. MB1 をクリックして選択する。「Help」ウィンドウ内に，そのオブジェクトまたは領域についてのヘルプが表示される。補助トピックによって，適切な場合はタスク用の説明が表示される。

ダイアログ・ボックスのコンテキスト依存のヘルプを表示するには，ダイアログ・ボックス内で「Help」ボタンをクリックして選択することもできます。

注意

第 12 章は作業別に構成されており，デバッガのヒープ・アナライザの使い方を説明しています。

「Stop」以外のプッシュ・ボタンでは，本当の意味でコンテキスト依存のヘルプを表示することはできません。これは，他のボタンがすべて，変更や削除を行えるようになっているためです。

8.4.2 「Overview」ヘルプ・トピックとサブトピックの表示

「Overview」ヘルプ・トピック(「Overview of the Debugger」)とそのサブトピックには，デバッガとデバッグに関するタスク用の情報が表示されます。

「Overview」トピックを表示するには，次のどちらかの方法を使用してください。

- デバッガのウィンドウの「Help」メニューで「On Window」を選択する。
- デバッガのヘルプ・ウィンドウの「View」メニューで「Go To Overview」を選択する。

「Overview」トピックを表示した後で特定のトピックを表示するには，参照したいトピックを補助トピックの一覧の中から選択します。

8.4.3 デバッガ・コマンドについてのヘルプの表示

デバッガ・コマンドについてのヘルプを表示するには，次のようにします。

1. デバッガのウィンドウの「Help」メニューで「Commands」を選択する。
2. 補助トピックの一覧からコマンド名またはその他のトピック(たとえば「Language_Support」)を選択する。

ヘルプ・コマンドはコマンド・ビューのコマンド行インタフェースにはありませんので，ご注意ください。

8.4.4 デバッガの診断メッセージについてのヘルプの表示

デバッガの診断メッセージはコマンド・ビューに表示されます。特定の診断メッセージについてのヘルプを表示するには，次の手順に従ってください。

1. デバッガのウィンドウの「Help」メニューで「Commands」を選択する。
2. 補助トピックの一覧から「Messages」を選択する。
3. 補助トピックの一覧からメッセージ識別子を選択する。

9
デバッグ・セッションの開始と終了

本章では次の方法について説明します。

- デバッガの起動 (第 9.1 節)
- プログラムの実行終了後のデバッグの続行 (第 9.2 節)
- 現在のデバッグ・セッションからの同一プログラムの再実行 (第 9.3 節)
- 現在のデバッグ・セッションからの別のプログラムの実行 (第 9.4 節)
- プログラムの実行に対する割り込みおよびデバッガ動作の強制終了 (第 9.6 節)
- デバッグ・セッションの終了 (第 9.7 節)
- 特定の用途の追加オプションによるデバッガの起動 (第 9.8 節)
- サブプロセスまたは独立プロセスとしてすでに動作中のプログラムのデバッグ (第 9.5 節)

9.1 保持デバッガの起動

この節では， DCL レベル($)からデバッガを起動して，ユーザのプログラムをデバッガの制御下に置く一般的な方法について説明します。オプションの起動方法については，第 9.8 節を参照してください。

ここで説明している方法に従って，保持デバッガでデバッガを起動すると， Connect (第 9.5 節を参照)， Rerun (第 9.3 節を参照)，および Run (第 9.4 節を参照) の機能を使用できるようになります。

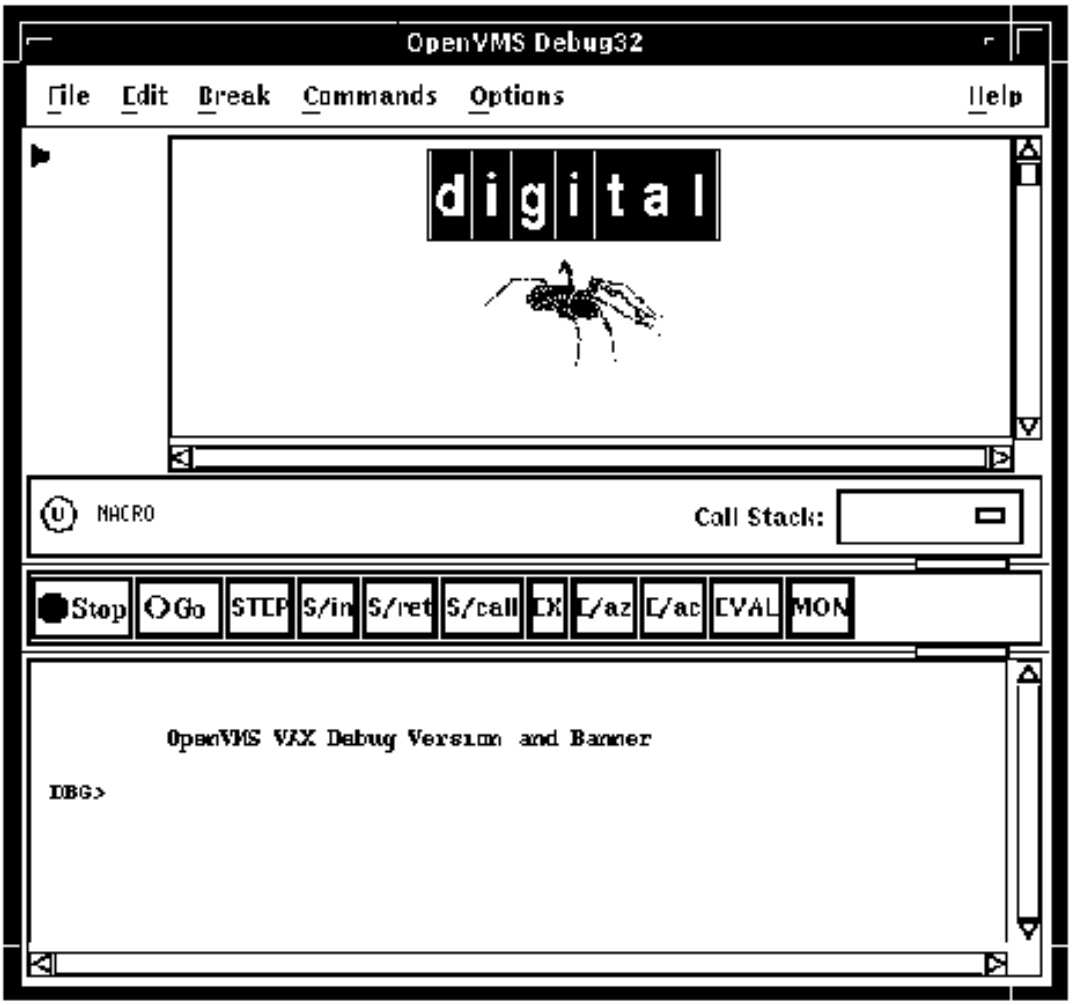
デバッガを起動してプログラムをデバッガの制御下に置くには，次のようにします。

1. 第 1.2 節で説明したとおりにプログラムをコンパイルおよびリンクしてあることを確認する。
2. 次のコマンド行を入力する。

   ```
   $ DEBUG/KEEP
   ```

   省略時の設定では，図 9–1 のようにデバッガが起動します。プログラムをデバッガの制御下に置く (手順 4) までメイン・ウィンドウは空のままです。デバッガの起動時にはユーザ定義初期化ファイルが実行されます (第 13.2 節を参照)。

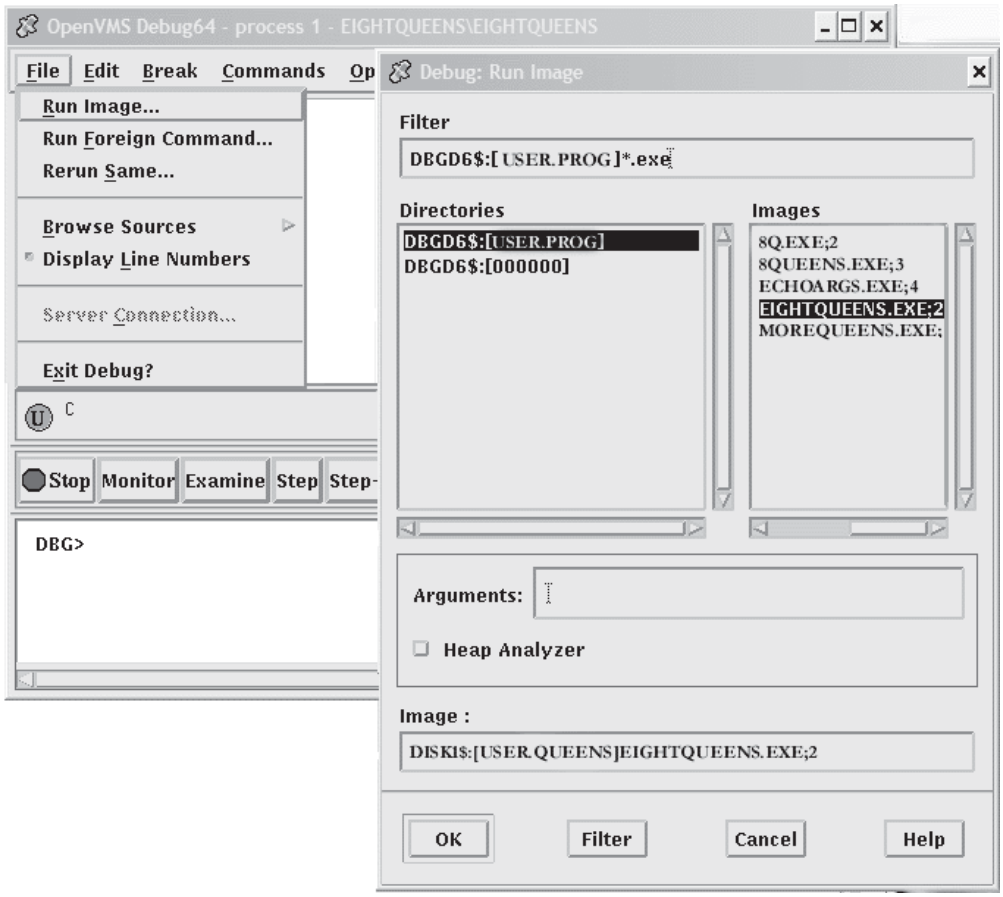
図 9–1 起動時のデバッガ



3. 次の 3 つのいずれかの方法を使用して，プログラムをデバッガの制御下に置く。
   - プログラムがサブプロセスまたは独立プロセスとしてすでに実行されている場合， CONNECT コマンドを使用してプログラムをデバッガの制御下に置きます。第 9.5 節を参照してください。
   - 指定のイメージの実行 (最も一般的な方法)
     1. メイン・ウィンドウの「File」メニューから「Run Image...」を選択する。「Run Image」ダイアログ・ボックスが現れ，ユーザのカレント・ディレクトリにある実行可能なイメージの一覧が表示される (図 9–2 を参照)。
     2. デバッグするイメージの名前をクリックして選択する。「Image:」フィールドにそのイメージ名が表示される。

3. プログラムに渡す引数がある場合は，「Arguments:」フィールドに入力する。デバッガは文字列の解析時に引用符を取り除くので，引用符付きの文字列の場合は二重引用符を追加しなければならない。
4. 「OK」をクリックする。

図 9–2 イメージの指定によるプログラムの実行

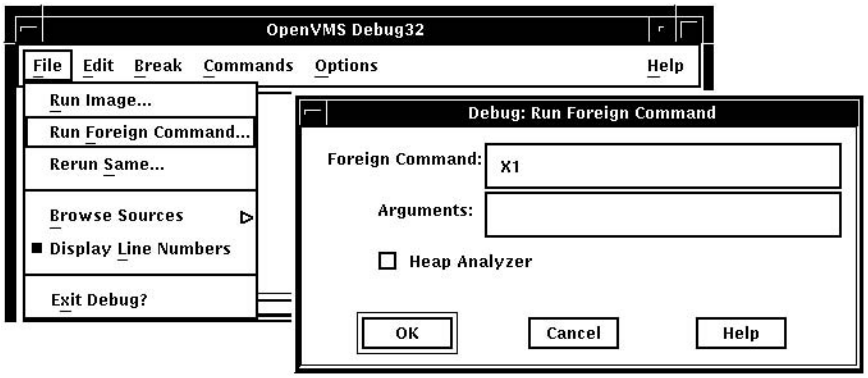


- DCL コマンドまたはフォーリン・コマンドのシンボルの指定によるイメージの実行
  1. メイン・ウィンドウの「File」メニューから「Run Foreign Command...」を選択する。「Run Foreign Command...」ダイアログ・ボックスが表示される (図 9–3 を参照)。
  2. 「Foreign Command:」フィールドにコマンドを入力する。このようなシンボルにより，ディレクトリ選択およびファイル選択のプロセスでショートカットを使用できるようになる。図 2-3 に表示されているフォーリン・コマンド X1 は，すでに次のように定義されている。

```
$X1 :== RUN MYDISK:[MYDIR.MYSUBDIR]EIGHTQUEENS.EXE
```

3. コマンドに付けて渡す引数を「Arguments:」フィールドに入力する。
4. 「OK」をクリックする。

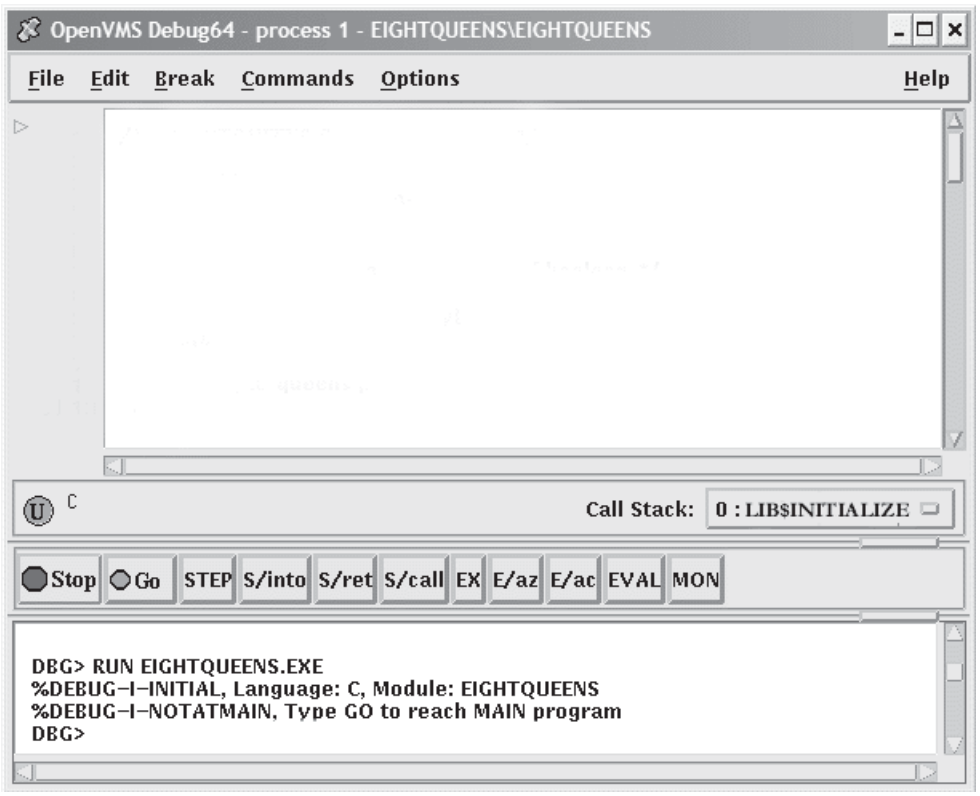
図 9–3 コマンド・シンボルの指定によるプログラムの実行



デバッガが，プログラムを制御できるようになると，デバッガは次の動作を行います。

- 図 9–4 に示すように，メイン・ウィンドウにプログラムのソース・コードを表示する。
- メイン・プログラムの先頭で実行を中断する。ソース・コードの左にある現在位置ポインタが指している行は，次に実行されるコードを表す。

図 9–4 起動時のソース・ディスプレイ



コマンド・ビューに表示されたメッセージは，このデバッグ・セッションが C プログラム用に初期化されており，ソース・モジュールの名前が EIGHTQUEENS であることを示しています。

ある種のプログラムでは，メイン・プログラムの前に，初期化コードの先頭でプログラムの実行が中断されるよう一時的なブレークポイントが設定され，次のメッセージが表示されます。

```
Type GO to reach MAIN program
No source line for address: nnnnnnnn
```

いくつかのプログラム (たとえば Ada) では，完全シンボル情報を使用して，ブレークポイントで初期化コードをデバッグすることができます。初期化を行うと，言語別のデバッガ・パラメータが設定されます。これらのパラメータは，デバッガが名前や式を解析する方法，デバッガが出力する形式などを制御します。

これにより，第 10 章の方法を使用してプログラムをデバッグできるようになります。

デバッガの制御下でのプログラムの実行については，次の制限事項に注意してください。

- 実行中のプログラムにデバッガを接続するために，この節で示した手順を使用することはできない (第 9.8.2 項を参照)。
- ネットワーク・リンクを通じて，デバッガの制御下でプログラムを実行するには，デバッガ・クライアント/サーバ・インタフェースを使用しなければならない。詳細は第 9.9 節を参照。

存在しないプログラムを実行しようとしたり，存在するプログラムの名前の綴りを誤って入力すると，次のエラー・メッセージがコマンド・ビューではなく， DECterm ウィンドウに表示されます。

```
%DCL-W-ACTIMAGE, error activating image
-CLI-E-IMAGEFNF, image file not found
```

9.2 プログラムの実行の終了

1 つのデバッグ・セッションの中でプログラムの実行が正常に終了すると，次のメッセージが発行されます。

このときユーザには次のオプションが与えられます。

- 同じデバッグ・セッションからプログラムを再実行することができる (第 9.3 節を参照)。
- 同じデバッグ・セッションから別のプログラムを実行することができる (第 9.4 節を参照)。
- デバッグ・セッションを終了することができる (第 9.7 節を参照)。

9.3 現在のデバッグ・セッションからの同一プログラムの再実行

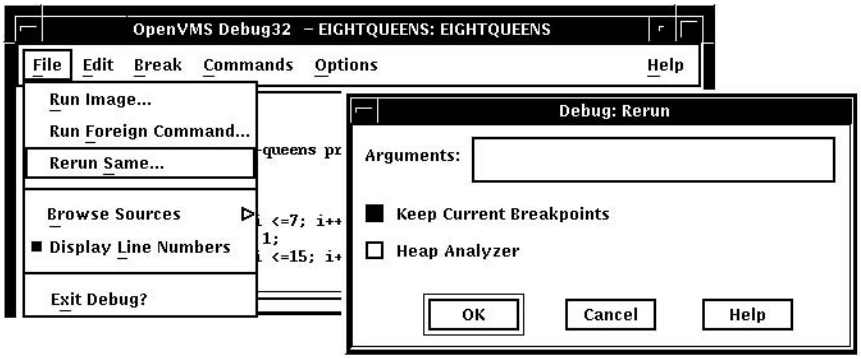
保持デバッガでデバッガを実行する場合 (第 9.1 節を参照)，デバッグ・セッションを実行している間ならいつでも，現在デバッガによって制御されているプログラムを再実行できます。

プログラムを再実行するには，次の手順に従ってください。

1. メイン・ウィンドウ「File」メニューから「Rerun Same...」を選択する。「Rerun」ダイアログ・ボックスが表示される (図 9–5 を参照)。
2. プログラムに渡す適当な引数が必要な場合は，「Arguments:」フィールドに入力する。デバッガは文字列の解析時に引用符を取り除くので，引用符付きの文字列の場合は二重引用符を追加しなければならない。

3. 以前に設定したか，有効または無効にしたブレークポイント，トレースポイント，または静的ウォッチポイントの現在の状態を保存するか保存しないかを選択する (第 10.4 節および第 10.5.5 項を参照)。非静的ウォッチポイントが保存されるか保存されないかは，ウォッチされる変数の有効範囲に応じて，実行の再開地点のメイン・プログラム・ユニットとの関連で決まる。
4. 「OK」をクリックする。

図 9–5 同一プログラムの再実行



プログラムを再実行するときの初期状態は，保存したブレークポイント，トレースポイント，静的ウォッチポイントを除き，第 9.1 節の説明に従ってプログラムをデバッガの制御下に置いた場合の初期状態と同じです。ソース表示と現在位置ポインタは適宜に更新されます。

プログラムを再実行する場合，デバッガは，現在デバッガの制御下にあるイメージと同じバージョンのイメージを使用します。同一のデバッグ・セッションからそのプログラムの別のバージョン (または他のプログラム) をデバッグするには，メイン・ウィンドウの「File」メニューから「Run Image...」または「Run Foreign Command..」を選択してください (第 9.1 節を参照)。

9.4 現在のデバッグ・セッションからの別のプログラムの実行

最初に第 9.1 節の説明に従ってデバッガを起動した場合は，1 つのデバッグ・セッションの中でいつでも別のプログラムをデバッガの制御下に置くことができます。プログラムをデバッガの制御下に置くには，第 9.1 節の手順に従ってください。その手順を使用するときの制限事項にも注意してください。

9.5 すでに実行中のプログラムのデバッグ

ここでは，サブプロセスまたは独立プロセスとしてすでに実行中のプログラムをデバッグする方法を説明します。次の手順を実行します。

1. DCL コマンドを使用して，保持デバッガの構成を開始します。

   ```
   $ DEBUG/KEEP
   ```

2. DBG>プロンプトで， CONNECT コマンドを使用してプログラムに割り込み，プログラムをデバッグ制御下に置きます。 CONNECT を使用すると，サブプロセスとして動作中のプログラムに接続したり，独立プロセスとして動作中のプログラムに接続することができます。独立プロセスは，次の要件の両方を満たさなければなりません。

   - 独立プロセスの UIC は，ユーザのプロセスと同じグループでなければならない。
   - 独立プロセスには， CLI が対応付けられていなければならない。

   2 番目の要件は，実際には，プログラムが次のようなコマンドで起動されていなければならないということです。

   ```
   $ RUN/DETACH/INPUT=xxx.com SYS$SYSTEM:LOGINOUT
   ```

   xxx.comは，プログラムを/NODEBUG で起動するコマンド・プロシージャです。

   いったんプログラムに接続すると，その後のデバッグ・セッションは，通常のデバッグ・セッションと同じです。

3. プログラムのデバッグが終了したら，次のいずれかの操作を行います。

   - DISCONNECT コマンドを使用して，プログラムをデバッガの制御から解放します。プログラムは実行を続けます。
   - デバッガを終了します。プログラムは終了します。

9.6 プログラムの実行に対する割り込みおよびデバッガ動作の強制終了

デバッグ・セッション中にプログラムの実行に割り込みをかけるには，プッシュ・ボタン・ビューの「Stop」ボタンをクリックします (図 8–3 を参照)。これは，たとえばプログラムが無限ループに陥った場合に役立ちます。

進行中のデバッガ動作を強制終了するには，「Stop」をクリックします。これは，たとえばデバッガが長いデータ・ストリームを表示している場合に役立ちます。

「Stop」をクリックしてもデバッグ・セッションは終了しません。プログラムが動作していないとき，またはデバッガがコマンドを実行していないときは「Stop」をクリックしても何も起こりません。

9.7 デバッグ・セッションの終了

デバッグ・セッションを終えるためデバッガを終了するには，メイン・ウィンドウの「File」メニューから「Exit Debugger」を選択するか，プロンプトで EXIT を入力します (確認ダイアログを回避する場合)。これでシステム・レベルに制御が戻ります。

現在のデバッグ・セッションからプログラムを再実行する方法については，第 9.3 節を参照してください。

現在のデバッグ・セッションから別のプログラムを実行する方法については，第 9.4 節を参照してください。

9.8 デバッガを起動するときの追加オプション

DCL レベル ($) からデバッガを起動するときは，第 9.1 節で説明した起動の手順に加え，次のオプションを使用することができます。

- デバッグするプログラムを DCL の RUN コマンドで実行することによって，デバッガを起動する (第 9.8.1 項を参照)。
- Ctrl/Y を押して，実行中のプログラムに割り込みをかけてから， DCL の DEBUG コマンドを使用してデバッガを起動する (第 9.8.2 項を参照)。
- 次を行うために，デバッガの省略時のインタフェース (DECwindows Motif for OpenVMS ユーザ・インタフェース) を無効にする (第 9.8.3 項を参照)。
  - 別のワークステーション上で DECwindows Motif for OpenVMS ユーザ・インタフェースを表示する。
  - コマンド・インタフェースをプログラムの入出力 (I/O) とともに DECterm Motif ウィンドウに表示する。
  - コマンド・インタフェースとプログラムの入出力 (I/O) を別々の DECterm ウィンドウに表示する。

どの場合もデバッガの起動前に，第 1.2 節の説明に従ってプログラム・モジュールをコンパイルおよびリンクした。

9.8.1 プログラムの実行によるデバッガの起動

DCL コマンドのRUN *program-image*を入力すると， 1 つの手順で，デバッガを起動してプログラムをデバッガの制御下に置くことができます。そのプログラムは/DEBUG 修飾子を使用してコンパイルとリンクが行われているものと想定されます。

しかし，この方法で起動した場合は，第 9.3 節と第 9.4 節でそれぞれ説明した再実行機能と実行機能を使用することはできません。デバッガの制御下で同一プログラムを再実行するか別のプログラムを実行するには，いったんデバッガを終了してからもう一度起動する必要があります。

プログラムの実行によってデバッガを起動するには， DCL コマンドのRUN *program-image*を入力してデバッガを起動します。次に例を示します。

```
$ RUN EIGHTQUEENS
```

省略時の設定では，デバッガが図 9–4 のように起動され，ユーザ定義初期化ファイルが実行され，メイン・ウィンドウにプログラムのソース・コードが表示されます。現在位置ポインタは，メイン・プログラムの先頭で実行を一時停止していることを示しています。そして，メイン・プログラム・ユニットのソース言語に合わせて言語固有のパラメータが設定されます。

デバッガの起動についての詳しい説明は，第 9.1 節を参照してください。

9.8.2 実行中のプログラムの割り込み後のデバッガの起動

ユーザは実行中のプログラムを自由にデバッガの制御下に置くことができます。これは，プログラムが無限ループに陥っていると思われるときや，出力が誤っていることに気付いた場合などに役立ちます。

プログラムをデバッガの制御下に置くには，次の手順に従ってください。

1. デバッガの制御の外でプログラムを実行するために， DCL コマンドのRUN/NODEBUG *program-image*を入力する。
2. 実行中のプログラムに割り込みをかけるために Ctrl/Y を押す。 DCL コマンド・インタプリタに制御が渡される。
3. デバッガを起動するために， DCL コマンドの DEBUG を入力する。

次に例を示します。

```
$ RUN/NODEBUG EIGHTQUEENS
   .
   .
   .
Ctrl/Y
Interrupt
$ DEBUG
[starts debugger]
```

デバッガの起動時には，メイン・ウィンドウが表示され，ユーザ定義初期化ファイルが実行されます。また，実行に割り込みがかけられたモジュールのソース言語に合わせて，言語固有のパラメータが設定されます。

どこで実行に割り込みがかけられたかを確認するには，次のようにします。

1. メイン・ウィンドウを見る。
2. コマンド入力プロンプトに SET MODULES/CALLS コマンドを入力する。
3. ソース・ウィンドウに「Call Stack」メニューを表示して，呼び出しスタック上のルーチン呼び出しの並びを確認する。レベル 0 のルーチンが，現在実行を一時停止されているルーチンである (第 10.3.1 項を参照)。

この方法でデバッガを起動した場合は，第 9.3 節と第 9.4 節でそれぞれ説明した再実行機能と実行機能を使用することはできません。デバッガの制御下で同一プログラムを再実行するか別のプログラムを実行するには，いったんデバッガを終了してからもう一度起動する必要があります。

デバッガの起動についての詳しい説明は，第 9.1 節を参照してください。

9.8.3 デバッガの省略時のインタフェースの変更

ワークステーションで DECwindows Motif for OpenVMS が稼動している場合，省略時の設定では，デバッガは DECwindows Motif for OpenVMS ユーザ・インタフェースで起動されます。 DECwindows Motif for OpenVMS ユーザ・インタフェースは， DECwindows Motif for OpenVMS のアプリケーション全体に通用する論理名 DECW$DISPLAY で指定されたワークステーションに表示されます。

ここでは，次の操作を行うためにデバッガの省略時の DECwindows Motif ユーザ・インタフェースを無効にする方法について説明します。

- デバッガの DECwindows Motif for OpenVMS ユーザ・インタフェースを別のワークステーション上に表示する。
- デバッガのコマンド・インタフェースをプログラムの入出力 (I/O) とともに DECterm ウィンドウに表示する。
- デバッガのコマンド・インタフェースとプログラムの入出力 (I/O) を別々の DECterm ウィンドウに表示する。

論理名 DBG$DECW$DISPLAY によって，デバッガの省略時のインタフェースを変更することができます。ほとんどの場合，省略時設定が適正なので DBG$DECW$DISPLAY を定義する必要はありません。

論理名 DBG$DECW$DISPLAY と論理名 DECW$DISPLAY については，第 9.8.3.4 項を参照してください。

9.8.3.1 別のワークステーション上でのデバッガの DECwindows Motif for OpenVMS ユーザ・インタフェースの表示

画面の大部分を使用する DECwindows Motif for OpenVMS アプリケーションをデバッグする場合 (または， Motif アプリケーションのポップアップをデバッグする場合)， 1 台のワークステーションでプログラムを実行し，別のワークステーションにデバッガの DECwindows Motif for OpenVMS ユーザ・インタフェースを表示すると便利です。その場合は次の手順に従ってください。

1. プログラムを実行しようとしている DECterm ウィンドウに，次の構文で論理定義を入力する。

   ```
   DEFINE/JOB DBG$DECW$DISPLAY workstation_pathname
   ```

   *workstation_pathname*は，デバッガの DECwindows Motif for OpenVMS ユーザ・インタフェースを表示するワークステーションのパス名です。このパス名の構文については，『OpenVMS DCL ディクショナリ』の SET DISPLAY コマンドの説明を参照してください。

   なるべくジョブ定義を使用してください。プロセス定義を使用する場合は，CONFINE 属性を与えないでください。

2. 論理定義を入力した DECterm ウィンドウからプログラムを実行する。DBG$DECW$DISPLAY で指定したワークステーション上にデバッガの DECwindows Motif for OpenVMS ユーザ・インタフェースが表示されます。アプリケーションのウィンドウ化インタフェースは，通常それが表示されるワークステーションに表示されます。

3. クライアント/サーバ・モードの使用 (第 9.9.2 項を参照)。

9.8.3.2 DECterm ウィンドウへのデバッガのコマンド・ユーザ・インタフェースの表示

デバッガのコマンド・インタフェースをプログラムの入出力 (I/O) とともに DECterm ウィンドウの中に表示するには，次の手順に従ってください。

1. デバッガを起動しようとしている DECterm ウィンドウに次の定義を入力する。

   ```
   $ DEFINE/JOB DBG$DECW$DISPLAY " "
   ```

   二重引用符の間には 1 つ以上のスペース文字を指定することができます。論理名にはジョブ定義を使用してください。プロセス定義を使用する場合は，CONFINE 属性を与えないでください。

2. 通常の方法で，その DECterm ウィンドウからデバッガを起動する (第 9.1 節を参照)。

   デバッガのコマンド・インタフェースが同一ウィンドウに表示されます。

次に例を示します。

```
$ DEFINE/JOB DBG$DECW$DISPLAY " "
$ DEBUG/KEEP

          Debugger Banner and Version Number

DBG>
```

これで，第 9.1 節で説明した方法でプログラムをデバッガの制御下に置くことができます。

9.8.3.3 別の DECterm ウィンドウへのコマンド・インタフェースとプログラムの入出力 (I/O) の個別表示

ここでは，デバッガを起動する DECterm ウィンドウ以外の DECterm ウィンドウに，デバッガのコマンド・インタフェースを表示する方法について説明します。画面用プログラムのデバッグにコマンド・インタフェースを使用する場合は，この別ウィンドウが便利です。

- プログラムの入出力 (I/O) (I/O) はデバッガを起動したウィンドウに表示される。
- 画面モードの表示を含め，デバッガの入出力 (I/O) は別ウィンドウに表示される。

DECwindows Motif for OpenVMS ではなく VWS が稼動しているワークステーションの DBG>プロンプトで SET MODE SEPARATE コマンドを入力しても同じ効果が得られます。(DECterm ウィンドウの中では SET MODE SEPARATE コマンドは無効です。)

「Debugger」という別のデバッガ・ウィンドウにデバッガのコマンド・インタフェースを表示する方法を次の例に示します。

1. Example 9–1 のように，コマンド・プロシージャ SEPARATE_WINDOW.COM を作成する。

Example 9–1 **コマンド・プロシージャ** SEPARATE_WINDOW.COM

(**次ページに続く**)

Example 9–1 (続き) コマンド・プロシージャ SEPARATE_WINDOW.COM

```
$ ! DECtermウィンドウからのSET MODE SEPARATEの効果をシミュレートする。
$ !
$ CREATE/TERMINAL/NOPROCESS -
   /WINDOW_ATTRIBUTES=(TITLE="Debugger",-
          ICON_NAME="Debugger",ROWS=40)-
   /DEFINE_LOGICAL=(TABLE=LNM$JOB,DBG$INPUT,DBG$OUTPUT)
$ ALLOCATE DBG$OUTPUT
$ EXIT
$ !
$ ! CREATE/TERMINAL/NOPROCESSコマンドは，プロセスなしで
$ ! DECtermウィンドウを作成する。
$ !
$ ! /WINDOW_ATTRIBUTES修飾子は，ウィンドウの
$ ! 名前(Debugger)，アイコン名(Debugger)，および
$ ! ウィンドウの行数(40)を指定する。
$ !
$ ! /DEFINE_LOGICAL修飾子は，ウィンドウに論理名DBG$INPUTと
$ ! 論理名DBG$OUTPUTを割り当てるので，
$ ! ウィンドウがデバッガの入出力（I/O）装置になる。
$ !
$ ! ALLOCATE DBG$OUTPUTコマンドは，デバッグ・セッションの
$ ! 終了時に別ウィンドウをオープンしたまま残す。
```

2. 次のようにコマンド・プロシージャを実行する。

   ```
   $ @SEPARATE_WINDOW
   %DCL-I-ALLOC, _MYNODE$TWA8: allocated
   ```

   SEPARATE_WINDOW.COM で指定されている属性の新しい DECterm ウィンドウが作成される。

3. デバッガのコマンド・インタフェースを表示するために，第 9.8.3.2 項の手順を実行する。新しいウィンドウにコマンド・インタフェースが表示される。

4. これでデバッガのウィンドウにデバッガ・コマンドを入力することができる。プログラムの入出力 (I/O) は，デバッガを起動した DECterm ウィンドウに表示される。

5. EXIT コマンドでデバッグ・セッションを終了すると，プログラムの入出力 (I/O) ウィンドウ内の DCL プロンプトに制御が戻るが，デバッガのウィンドウは開いたまま残る。

6. 第 9.8.3.2 項のようにプログラムの入出力 (I/O) と同じウィンドウにデバッガのコマンド・インタフェースを表示するには，次のコマンドを入力する。

   ```
   $ DEASSIGN/JOB DBG$INPUT
   $ DEASSIGN/JOB DBG$OUTPUT
   ```

   ユーザが明示的に閉じるまで，デバッガのウィンドウは開いたままで ある。

9.8.3.4 DBG$DECW$DISPLAY と DECW$DISPLAY の説明

ワークステーションで DECwindows Motif for OpenVMS が稼動している場合，省略時の設定では，デバッガは DECwindows Motif for OpenVMS ユーザ・インタフェースで起動されます。 DECwindows Motif for OpenVMS ユーザ・インタフェースは， DECwindows Motif for OpenVMS のアプリケーション全体に通用する論理名 DECW$DISPLAY で指定されたワークステーションに表示されます。DECW$DISPLAY は， FileView または DECterm によってジョブ・テーブルの中に定義されます。 DECW$DISPLAY はワークステーション用の表示装置を指します。

DECW$DISPLAY についての詳しい説明は，『OpenVMS DCL ディクショナリ』の DCL コマンドの SET DISPLAY および SHOW DISPLAY の説明を参照してください。

論理名 DBG$DECW$DISPLAY は， DECW$DISPLAY と等価なデバッガ固有の論理名です。 DBG$DECW$DISPLAY は，デバッガ固有の論理名である DBG$INPUT と DBG$OUTPUT に相当します。これらの論理名を使用すると，それぞれ SYS$INPUT と SYS$OUTPUT に割り当てられた値を変更して，デバッガの入出力 (I/O) が表示される装置を指定することができます。

デバッガの省略時のユーザ・インタフェースが使用されるのは，DBG$DECW$DISPLAY が未定義である場合，または変換された DBG$DECW$DISPLAY が DECW$DISPLAY と同じである場合です。省略時の設定では DBG$DECW$DISPLAY は未定義です。

DECW$DISPLAY と DBG$DECW$DISPLAY の論理定義を使用している場合，デバッガは次のアルゴリズムに従って動作します。

1. 論理名 DBG$DECW$DISPLAY が定義されているときはそれを使用する。定義されていないときは DECW$DISPLAY 論理名を使用する。
2. 論理名を変換する。論理名の値が NULL でない場合 (文字列にスペース以外の文字が含まれている場合) は，指定されているワークステーション上に DECwindows Motif for OpenVMS ユーザ・インタフェースを表示する。論理名の値が NULL である場合 (文字列にスペースだけしか含まれていない場合) は，DECterm ウィンドウの中にコマンド・インタフェースを表示する。

OpenVMS デバッガが DECwindows Motif for OpenVMS ユーザ・インタフェースで起動するように設定するときは，次のいずれかのコマンドを使用します。

```
$DEFINE DBG$DECW$DISPLAY "WSNAME::0"
$SET DISPLAY/CREATE/NODE=WSNAME
```

ただしWSNAMEは，ワークステーションのノード名になります。

9.9 Motif デバッグ・クライアントの起動

OpenVMS デバッガ・バージョン 7.2 の機能であるクライアント/サーバ・インタフェースを使用すると， OpenVMS (VAX または Alpha CPU) 上で実行されているプログラムを，同じシステム上，または別のシステム上で実行されているクライアント・インタフェースからデバッグすることができます。

デバッガのクライアント/サーバでは，保持デバッガの機能をそのまま使用することができますが，デバッガは，デバッグ・サーバとデバッグ・クライアントという 2 つの構成要素に分割されています。デバッグ・サーバは OpenVMS システム上で実行され，ユーザ・インタフェースを持たない保持デバッガに相当します。一方，デバッグ・クライアントはユーザ・インタフェースを持ち， DECwindows Motif for OpenVMS を使用する OpenVMS システム上，または Microsoft Windows 95 か Microsoft Windows NT を使用する PC 上で実行されます。

9.9.1 ソフトウェアの必要条件

デバッグ・サーバの実行には， OpenVMS バージョン 7.2 以降が必要です。

デバッグ・クライアントは，次のいずれかのオペレーティング・システム上で実行できます。

- OpenVMS バージョン 7.2 以降 (DECwindows Motif for OpenVMS バージョン 1.2-3 が必要)
- Microsoft Windows 95
- Microsoft Windows NT バージョン 3.51 以降 (Intel または Alpha)

また， OpenVMS デバッガ・クライアント/サーバ構成では，サーバを実行する OpenVMS ノードに，次のものがインストールされていることが必要です。

- TCP/IP スタック
- DCE RPC

注意

TCP/IP Services for OpenVMS (UCX) バージョン 4.1 を実行している場合は，ECO2 がインストールされていることも必要になります。 UCX の最新バージョンを実行することもできます。

OpenVMS バージョン 7.2 のインストール・プロシージャでは，DCE RPC が自動的にインストールされます。

9.9.2 サーバの起動

OpenVMS システムに直接ログインしてからデバッグ・サーバを起動することもできますが， eXcursion のような製品や Telnet のようなターミナル・エミュレータを使用して，リモートでログインするほうが便利です。

デバッグ・サーバを起動するには，次のコマンドを入力します。

```
$ DEBUG/SERVER
```

サーバのネットワーク・バインド文字列が表示されます。サーバのポート番号は，角括弧 ([ ]) で囲まれて表示されます。例を示します。

```
$ DEBUG/SERVER

%DEBUG-I-SPEAK: TCP/IP: YES, DECnet: YES, UDP: YES
%DEBUG-I-WATCH: Network Binding: ncacn_ip_tcp:16.32.16.138[1034]
%DEBUG-I-WATCH: Network Binding: ncacn_dnet_nsp:19.10[RPC224002690001]
%DEBUG-I-WATCH: Network Binding: ncadg_ip_udp:16.32.16.138[1045]
%DEBUG-I-AWAIT: Ready for client connection...
```

クライアントから接続する場合は，サーバを指定するために，いずれかのネットワーク・バインド文字列を使用します (第 9.9.4 項を参照)。次の表に，ネットワーク・トランスポートとネットワーク・バインド文字列の接頭辞の対応を示します。

ネットワーク・トランスポート	ネットワーク・バインド文字列の接頭辞
TCP/IP	ncacn_ip_tcp
DECnet	ncacn_dnet_nsp
UDP	ncadg_ip_udp

注意

通常は，ノード名とポート番号だけを使用してサーバを指定することができます。nodnam[1034]**がその例です。**

省略時の設定では，サーバを起動したウィンドウに，メッセージとプログラム出力が表示されます。必要に応じて，プログラム出力を別のウィンドウにリダイレクトすることができます。

次の例には， DCE がインストールされていないことを示すエラー・メッセージが含まれています。

```
$ debug/server
%LIB-E-ACTIMAGE, error activating image disk:[SYSn.SYSCOMMON.][SYSLIB]DTSS$SHR.EXE;
-RMS-E-FNF, file not found
```

このエラー・メッセージの場合， DCE はインストールされていますが，設定されていません。

9.9.3 プライマリ・クライアントとセカンダリ・クライアント

デバッガのクライアント/サーバ・インタフェースを使用すると，同じサーバに 1 つ以上のクライアントを接続することが可能になります。この機能を利用すると，チームによるデバッグや教室でのセッション，その他のアプリケーションを実現できます。

プライマリ・クライアントは，サーバに最初に接続されたクライアントです。セカンダリ・クライアントは，同じサーバに後から接続されたクライアントです。プライマリ・クライアントによって，サーバにセカンダリ・クライアントを接続できるようにするかどうかが制御されます。

セッション中で，いくつのセカンダリ・クライアントを使用できるようにするかを指定する方法については，第 9.9.4 項で説明します。

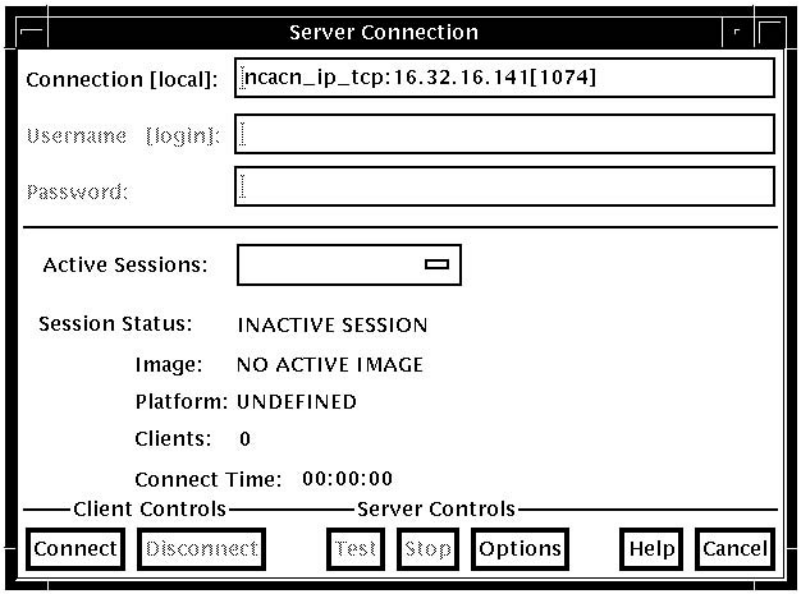
9.9.4 Motif クライアントの起動

セッションとは，特定のクライアントと特定のサーバの間の接続のことです。クライアントでは，サーバへの接続の際にクライアントが使用したネットワーク・バインド文字列によって，各セッションが識別されます。デバッグ・サーバを起動してから，Motif デバッガ・クライアントを起動してください。起動するには，次のコマンドを入力ます。

```
$ DEBUG/CLIENT
```

Motif デバッグ・クライアントからセッションを確立するには，「File」メニューの「Server Connection」をクリックします。「Server Connection」ダイアログの「Connection」リストには，省略時の設定のネットワーク・バインド文字列が表示されます。表示される文字列は直前に入力したものか，またはクライアントが実行されているノードとなります。サーバを省略時の設定のバインド文字列に関連付ける必要はありません。図 9–6 に「Server Connection」ダイアログを示します。

図 9–6 「Server Connection」ダイアログ



「Server Connection」ダイアログの下部にあるボタンを使用すると，次のことを行えます。

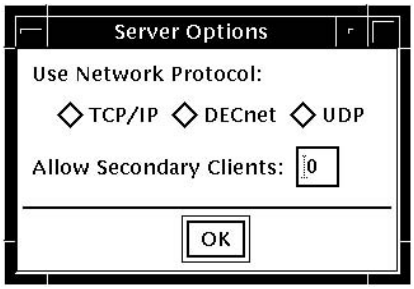
- 選択したサーバに接続して，新規のセッションを起動する。
- セッションを切断する。
- セッションがまだ使用可能かどうかをテストする。
- サーバを停止する。
- 接続操作を取り消して，ダイアログを消去する。

また，「Options」ボタンをクリックすると，「Server Options」ダイアログが表示されます。「Server Options」ダイアログでは，使用するトランスポートを選択することができます (第 11.5.1 項を参照)。

「Server Options」ダイアログでは，新規のセッションで使用できるセカンダリ・クライアントの数 (0 ～ 31) を選択することもできます。

図 9–7 に「Server Options」ダイアログを示します。

図 9–7 「Server Options」ダイアログ



クライアントをサーバに接続するには，次の手順に従ってください。

1. 「File」メニューを開く。
2. 「Server Connection」をクリックする。
3. 「Connection」フィールドに，サーバのネットワーク・バインド文字列を入力するか，省略時の設定の文字列のままにする。
4. 「Options」をクリックする。
5. 「Server Options」ダイアログで，ネットワーク・トランスポートをクリックする (TCP/IP， DECnet，または UDP)。
6. 「Server Options」ダイアログで，使用できるようにするセカンダリ・クライアントの数を選択する (0 ～ 31)。
7. 「OK」をクリックして，「Server Options」ダイアログを消去する。
8. 「Server Connection」ダイアログで，「Connect」をクリックする。

上記の手順を繰り返し，新規のネットワーク・バインド文字列をその都度指定することによって，いくつでも，サーバに対する接続を確立することができます。

9.9.5 セッションの切り替え

サーバに接続してセッションを起動すると，「Server Connection」ダイアログの「Active Sessions」リストにセッションが表示されます (図 9–8 を参照)。前後のセッションに切り替えることが可能です。新規のセッションに切り替えると，表示されているデバッガ画面の内容が，デバッガによって新しい内容に更新されます。

別のセッションに切り替えるには，次の手順に従ってください。

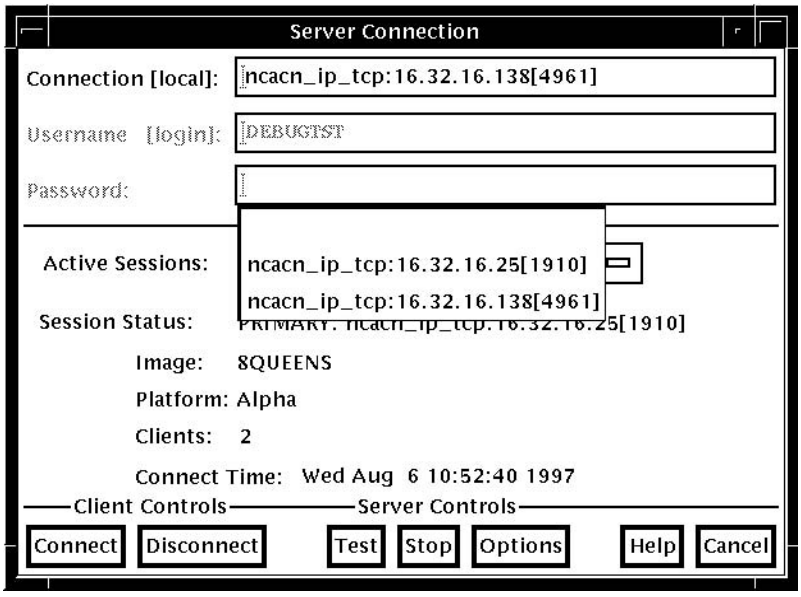
1. 「File」メニューを開く。
2. 「Server Connection」をクリックする。
3. 「Active Sessions」リストをクリックして，アクティブなセッションのリストを表示する。

4. 「Active Sessions」リスト内の目的のセッションをダブルクリックする。ダブルクリックしたセッションがカレント・セッションとして選択されて，「Server Connection」ダイアログが消去されて，デバッガ画面が現在の内容に更新される。

セッションがアクティブになっている間は，セッションで使用できるセカンダリ・クライアントの数を変更できない点に注意してください。セッションで使用できるセカンダリ・クライアントの数を変更するには，プライマリ・クライアントで次の手順に従ってください。

1. 「File」メニューを開く。
2. セッションのネットワーク・バインド文字列を指定する。
3. 「Disconnect」をクリックする。
4. 「Options」をクリックする。
5. 「Server Options」ダイアログで，ネットワーク・トランスポートをクリックする (TCP/IP， DECnet，または UDP)。
6. 「Server Options」ダイアログで，使用できるようにするセカンダリ・クライアントの数を選択する (0 ～ 31)。
7. 「OK」をクリックして，「Server Options」ダイアログを消去する。
8. 「Server Connection」ダイアログで，「Connect」をクリックする。

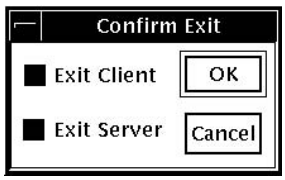
図 9–8 「Active Sessions」リスト



9.9.6 クライアント/サーバ・セッションの終了

「File」メニューの「Exit Debug?」をクリックして，「Confirm Exit」ダイアログを表示します。図 9–9 に「Confirm Exit」ダイアログを示します。

図 9–9 「Confirm Exit」ダイアログ



「Confirm Exit」ダイアログが表示されたら，次のいずれかを行ってください。

- クライアントとサーバの両方を終了する場合は，「OK」をクリックする (省略時の設定)。
- 何も行わずに「Confirm Exit」ダイアログを消去するには，「Cancel」をクリックする。
- デバッグ・クライアントのみを終了する場合は，次の手順に従う。
  1. 「Exit Server」をクリックする。
  2. 「OK」をクリックする。
- デバッグ・サーバのみを終了する場合は，次の手順に従う。
  1. 「Exit Client」をクリックする。
  2. 「OK」をクリックする。

デバッグ・サーバを終了しない場合は，別のデバッグ・クライアントからサーバに接続することが可能です。クライアントを終了しない場合は，ネットワーク・バインド文字列が分かっている別のサーバに接続することができます。

10

デバッガの使用方法

本章では次の操作方法について説明します。

- ユーザ・プログラムのソース・コードの表示 (第 10.1 節)
- デバッガ制御下のユーザ・プログラムの編集 (第 10.2 節)
- デバッガの制御下でのユーザ・プログラムの実行 (第 10.3 節)
- ブレークポイントによる実行の中断 (第 10.4 節)
- プログラム変数の検査と操作 (第 10.5 節)
- プログラム変数へのアクセス (第 10.6 節)
- レジスタ値の表示と変更 (第 10.7 節)
- ユーザ・プログラムのデコード済み命令ストリームの表示 (第 10.8 節)
- タスキング・プログラムのデバッグ (第 10.9 節)
- デバッガの DECwindows Motif for OpenVMS ユーザ・インタフェースのカスタマイズ (第 10.10 節)

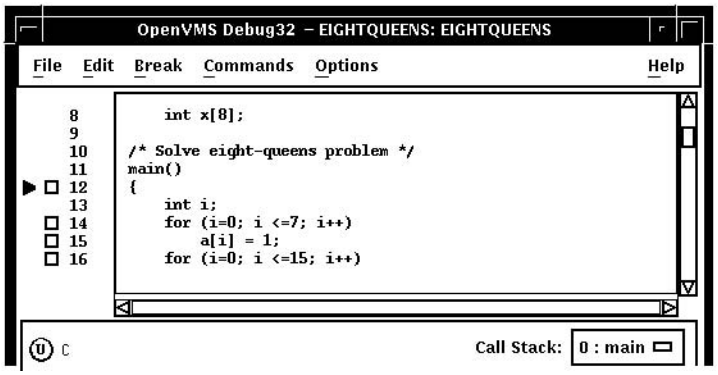
本章ではウィンドウの動作とウィンドウ・メニューの選択項目について説明しますが，一般的なデバッガ操作の大部分は，コンテキスト依存のポップアップ・メニューから項目を選択して実行できます。コンテキスト依存のポップアップ・メニューにアクセスするには，マウス・ポインタをウィンドウ領域に置いて MB3 をクリックします。

DECwindows Motif for OpenVMS のコマンド・プロンプトにコマンドを入力することもできます。デバッガ・コマンドの入力については，第 8.3 節を参照してください。この章で紹介しているプログラム EIGHTQUEENS.EXE および 8QUEENS.EXE のソース・コードについては，付録 D を参照してください。

10.1 ユーザ・プログラムのソース・コードの表示

デバッガはユーザ・プログラムのソース・コードをメイン・ウィンドウに表示します (図 10–1 を参照)。

図 10–1 ソース・ディスプレイ



実行が中断すると (たとえばブレークポイントで), デバッガはソース・ウィンドウを更新して，実行の停止箇所近くのコードを表示します。ソース・コードの左にある現在位置ポインタは，コードのどの行が次に実行されるかを示します。 1 行のソース行は，言語とコーディング方法によって異なりますが， 1 つまたは複数のプログラミング言語の文に相当します。

省略時の設定では，ソース・コードの左にコンパイラ生成行番号が表示されます。この行番号により，ブレークポイント・ビュー (第 10.4.4 項を参照) に表示されるブレークポイントを識別できます。ウィンドウに少しでも多くのソース・コードを表示したいときは，行番号を表示しないように選択することができます。行番号を表示するか表示しないかを指定するには，メイン・ウィンドウの「File」メニューで「Display Line Numbers」を選択します。

「Call Stack」メニューは，ソース・ビューとプッシュ・ボタン・ビューの間にありますが，ここには，表示されているソース・コードのルーチン名が表示されます。

現在位置ポインタは，図 10–1 に示されるように通常塗りつぶされています。表示コードが実行停止ルーチンのコードでない場合，現在位置ポインタは白抜きになります (第 10.1.3 項と第 10.6.2 項を参照)。

スクロール・バーを使用して，ソース・コードを次々と表示することができます。ただし，一度にスクロールできるのはユーザ・プログラムの 1 つの**モジュール**だけの，上下スクロールです。通常， 1 つのコンパイル単位が 1 つのモジュールです。多くのプログラミング言語では， 1 つのモジュールは 1 つのソース・ファイルの内容と対応しています。言語の中には Ada のように， 1 つのソース・ファイルの中に 1 つ以上のモジュールを含むものもあります。

次の各項では，プログラムの他の部分のソース・コードを表示する方法について説明します。したがって，各種のモジュールにブレークポイントを設定することなどができます。第 10.1.3 項では，表示するソース・コードが見つからない場合の処置について説明します。第 10.6.2 項では，呼び出しスタックで現在アクティブなルーチンのソース・コードの表示方法について説明します。

メイン・ウィンドウでの操作が終了したら，「Call Stack」メニューをクリックして，実行の停止箇所を表示し直すことができます。

コンパイル時にプログラムを最適化すると，表示されるソース・コードとプログラム記憶位置の実際の内容とが対応しなくなります(第 1.2 節を参照)。

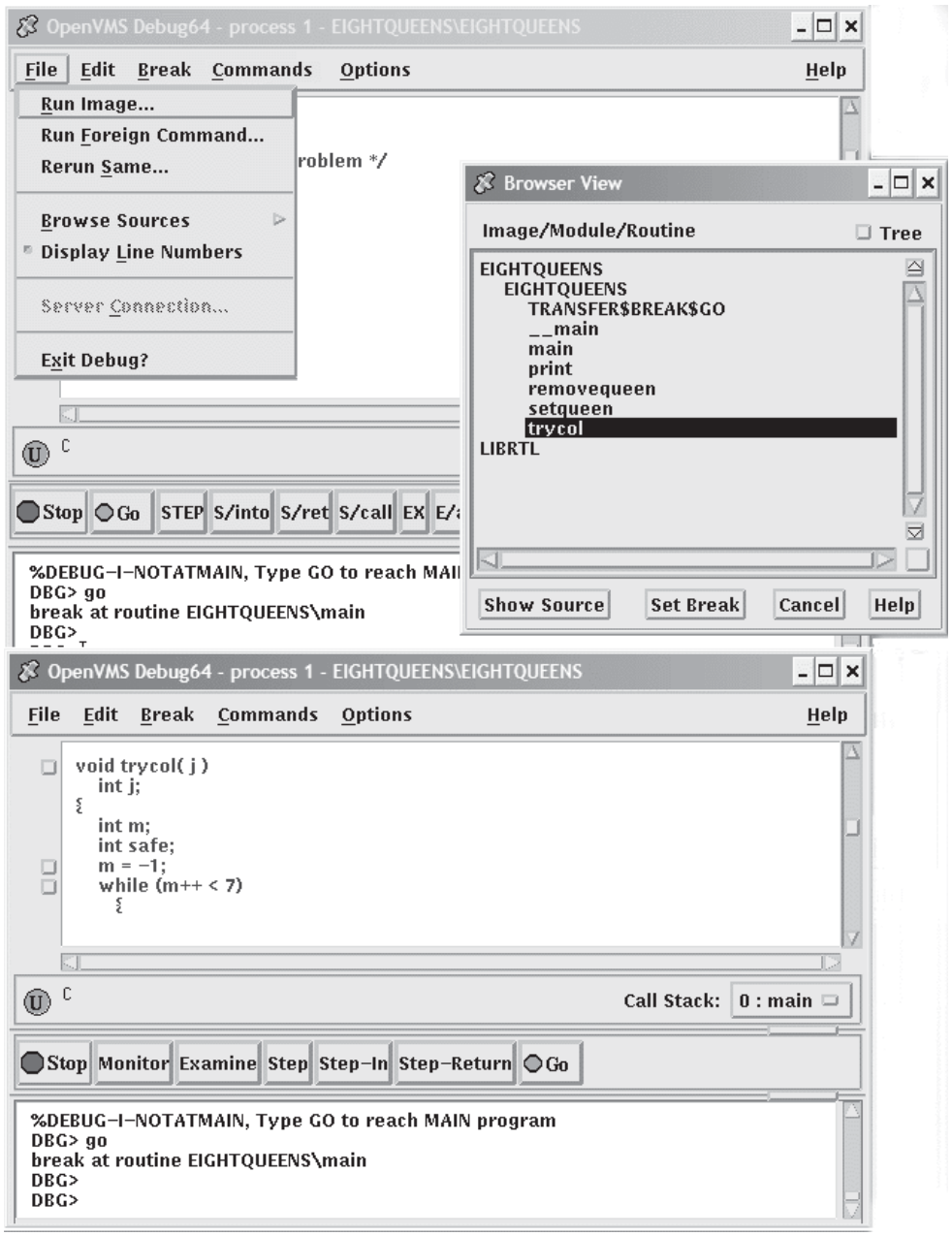
10.1.1 別ルーチンのソース・コードの表示

別ルーチンのソース・コードを表示するには，次の手順に従ってください。

1. メイン・ウィンドウの「File」メニューで「Browse Source...」を選択する (図 10–2 を参照)。「Source Browser」ダイアログ・ボックスに，実行可能なイメージおよびこれにリンクされたすべての共用可能イメージ (たとえば， DEBUG と LIBRTL) の名前が表示される。このとき実行可能なイメージは強調表示される。このイメージにシンボリック情報がない場合，リンクされたイメージの名前が薄く表示される。
2. 実行可能なイメージの名前をダブル・クリックする。そのイメージ名の下にインデントされて，そのイメージ内の各モジュールの名前が表示される。
3. 表示したいルーチンが含まれているモジュールの名前をダブル・クリックする。そのモジュール名の下にインデントされて，そのモジュール内の各ルーチンの名前が表示される。「Display Source」ボタンが強調表示される。
4. ソース・コードを表示したいルーチンの名前をクリックする。
5. 「Display Source」プッシュ・ボタンをクリックする。ルーチンのソース・コードがソース・ビューに表示され，同時にソース・コードの左に空のブレークポイント・ボタンが表示される。命令ビューがオープンしている場合，この表示が更新され，そのルーチンの機械語コードが表示される。

第 10.6.2 項では，呼び出しスタック内にあり，現在アクティブなルーチンのソース・コードを表示する別の方法を説明しています。

図 10–2 別ルーチンのソース・コードの表示



10.1.2 別モジュールのソース・コードの表示

別モジュールのソース・コードを表示するには，次の手順に従ってください。

1. メイン・ウィンドウの「File」メニューで「Browse Source...」を選択する。
「Source Browser」ダイアログ・ボックスに，実行可能なイメージおよびこれにリンクされたすべての共用可能イメージ(たとえば， DEBUG と LIBRTL) の名前

が表示される。このとき実行可能なイメージは強調表示される。このイメージにシンボリック情報がない場合，共有可能なイメージの名前が薄く表示される。

2. 実行可能なイメージの名前をダブル・クリックする。そのイメージ名の下にインデントされて，そのイメージ内の各モジュールの名前が表示される。
3. 表示したいソース・コードを含んでいるモジュールの名前をクリックする。「Display Source」ボタンが強調表示される。
4. 「Display Source」をクリックする。メイン・ウィンドウのソース表示にそのルーチンのソース・コードが表示される。命令ビューの命令ディスプレイをオープンしている場合，この表示が更新され，そのルーチンの命令コードが表示される。

10.1.3 目的のソース・コードを表示できない場合

ソース・コードを表示できない場合には，次のような原因が考えられます。

- デバッグ・オプションを指定しないでリンクまたはコンパイルされた
- シンボリック情報が得られないシステム・ルーチンやライブラリ・ルーチンの中で実行が一時停止する。そのような場合，プッシュ・ボタン・ビューで「S/Ret」ボタンを何回かクリックすれば，その呼び出し元ルーチンの実行にすぐ戻ることができる (第 10.3.5 項を参照)。
- コンパイル後にソース・ファイルが別のディレクトリに移動されている。第 10.1.4 項で，ソース・ファイルの場所をデバッガに指示する方法について説明する。

表示するソース・コードが見つからない場合，呼び出しスタックにある次のルーチンのソース・コードの表示が試行されます。そのようなルーチンのソース・コードが表示される場合，現在位置ポインタは，戻り先の呼び出し元ルーチンのソース行を示すために移動します。

10.1.4 ソース・ファイルの記憶位置の指定

ソース・ファイルの特性と記憶位置についての情報は，プログラムのデバッグ・シンボル・テーブルに入っています。ソース・ファイルがコンパイル後に別のディレクトリに移動された場合，そのソース・ファイルが見つからないことがあります。ソース・ファイルの記憶位置をデバッガに指示するには， DBG>プロンプトで SET SOURCE コマンドを入力します (第 8.3 節を参照)。

10.2 ユーザ・プログラムの編集

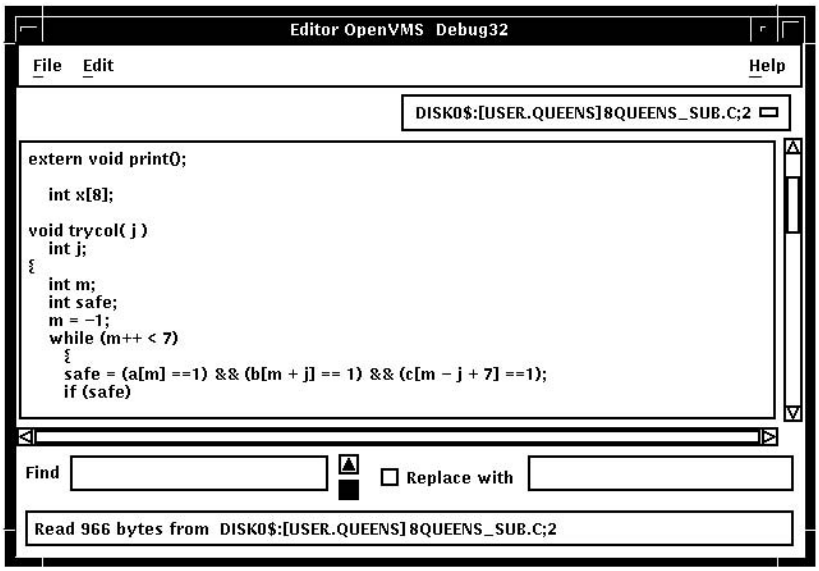
OpenVMS デバッガには簡単なテキスト・エディタが用意されており，ユーザ・プログラムをデバッグ中にソース・ファイルを編集することができます (図 10–3 を参照)。

デバッガの DECwindows Motif for OpenVMS メニュー・インタフェースで使用できるテキスト・エディタは，簡単な機能のエディタであるため，ランゲージ・センシティブ・エディタ (LSE) のような優れた機能を持つエディタにはおよびません。ただし「Commands」メニューの「Edit File」項目で起動するテキスト・エディタに，このような優れた機能を持つエディタを使用することはできません。内蔵エディタ以外のエディタを使用する場合は，コマンド・ビューの DBG>プロンプトで Edit コマンドを入力します (本書の第 3 部にある EDIT コマンドの項を参照)。

注意

コマンド・プロンプトに対して EDIT コマンドを入力すると，デバッガはデバッグ・セッションを起動した DECterm ウィンドウをユーザ定義エディタ・ウィンドウとして使用します (COMMANDS EDIT FILE プルダウン・メニューにハードワイヤ接続されているデバッガの組み込みエディタは使用されません)。この動作により，エディタを柔軟に選択できるようになります。 FILE EXIT または MWM Close を使用して，この DECterm ウィンドウを誤って終了した場合には，デバッグ・セッションは異常終了し，親ウィンドウが失われます。

図 10–3 エディタ・ウィンドウ



エディタを起動するには，メイン・ウィンドウの「Command」メニューで「Edit File」を選択します。省略時の設定では，このエディタによりバッファが 1 つオープ

ンされ，ソース・ビューに現在表示されているモジュールがそのバッファに表示されます。このバッファには，バッファのファイルに関するファイル指定の名前がつけられます。ソース・ビューにファイルが表示されない場合， main_buffer という名前を持つ空のテキスト・バッファが表示されます。バッファ名は，エディタ・ビューのメニュー・バーのすぐ下にあるバッファ・メニューに表示されます。

「File」メニューで「New」(空のテキスト・バッファ) または「Open」(既存のファイル) を選択すると，テキスト・バッファをいくつでも作成することができます。各テキスト・バッファの名前はバッファ・メニューに表示されます。バッファ間でテキストのカット，コピー，ペーストを実行するには，「Edit」メニューで項目を選択してから，バッファ・メニューでバッファを選択します。

前方検索，後方検索，置換の各操作を実行するには，「Find」と「Replace with」の各フィールドに文字列を入力してから，上下の方向を表した矢印をクリックします。Return キーを繰り返し押すと，文字列が繰り返し検索されます。「Edit」メニューで「Find/Replace Next」または「Find/Replace Previous」を選択して検索を繰り返すこともできます。

ファイルを保存するときは，「File」メニューから「Save」または「Save As」を選択します。変更したバッファをクローズしたり，デバッガを終了したりする前に，その内容を保存していない場合は，警告メッセージが表示されます。

ソース・コードを変更して，その結果をテストするときは，次の手順で行います。

1. デバッガを実行していないい DECterm ウィンドウを選択する。
2. プログラムをコンパイルし直す。
3. プログラムをリンクし直す。
4. デバッグ・セッションに戻る。
5. メイン・ウィンドウの「File」メニューから「Run Image...」を選択する。

10.3 プログラムの実行

この節では次の 3 つの内容について説明します。

- プログラムの実行停止箇所の特定
- プログラム実行の開始または再開
- プログラムのソース行の 1 行ずつの実行

現在のデバッグ・セッションで自分のプログラムを再実行したり別のプログラムを実行したりする方法については，第 9.3 節と第 9.4 節を参照してください。

10.3.1 実行の停止箇所の特定

プログラムの実行が一時停止している箇所を明らかにするには，次の手順に従ってください。

1. 現在位置ポインタがソース・ウィンドウに表示されていない場合，ソース・ウィンドウの「Call Stack」メニューをクリックして，現在位置ポインタを表示する (図 10–1 を参照)。
2. 現在位置ポインタを見る。
   - 塗りつぶしポインタの場合，そのポインタが指しているソース行のコードが次に実行される (第 10.1 節を参照)。この場合「Call Stack」メニューには，常に有効範囲レベル 0 のルーチン (実行の停止中) が示される。
   - 白抜きのポインタの場合，表示コードは呼び出し元ルーチンのソース・コードであり，ポインタは戻り先の呼び出し元ルーチン内のソース行を示している。
     - 「Call Stack」メニューでレベル 0 が示されている場合，実行が一時停止しているルーチンのソース・コードの表示はできない (第 10.1.3 項を参照)。
     - 「Call Stack」メニューで 0 以外のレベルが表示されている場合，呼び出し元ルーチンのソース・コードが表示されている (第 10.6.2 項を参照)。

呼び出しスタック上で現在アクティブなルーチン呼び出しの並びの一覧を表示するには，「Call Stack」メニューをクリックします。レベル 0 は実行が一時停止しているルーチンを示し，レベル 1 は呼び出し元ルーチンを示します。

10.3.2 プログラム実行の開始または再開

現在位置からプログラムの実行を開始したり実行を再開したりするには，プッシュ・ボタン・ビューで「Go」ボタンをクリックします (図 8–3 を参照)。

次のような状況では，デバッガの介入なしに自由にプログラムを実行するのが便利です。

- 無限ループの有無をテストする場合。この場合は，まず実行を開始する。プログラムが終了しないのでループしていると思われる場合は，「Stop」ボタンをクリックする。メイン・ウィンドウにはユーザが割り込んだ箇所が表示され，「Call Stack」メニューにはその箇所でのルーチン呼び出しの並びが示される (第 10.3.1 項を参照)。
- プログラムを特定の記憶位置まで直接実行する場合。この場合は，その記憶位置にブレークポイントを設定してから (第 10.4 節を参照)，実行を開始する。

プログラムを開始すると，次のいずれかが発生するまで実行が続きます。

- プログラムの実行が完了する。

- ブレークポイントに到達する (条件が真である条件付きブレークポイントを含む)。
- ウォッチポイントが検出される。
- 例外がシグナル通知される。
- ユーザがプッシュ・ボタン・ビューの「Stop」ボタンをクリックする。

プログラムの実行が中断すると，メイン・ウィンドウの表示が更新され，現在位置ポインタは次に実行されるコードの行を示します。

10.3.3 プログラムのソース行の 1 行ずつの実行

プログラムのソース行を 1 行ずつ実行するには，プッシュ・ボタン・ビューで「STEP」ボタンをクリックするか，またはコマンド・ビューで STEP コマンドを入力します。このデバッグ方法 (**ステップ実行**と呼ぶ) はよく使用されます。

ソース行が 1 行実行されると，ソース・ビューが更新され，現在位置ポインタは次に実行される行を示します。

ソース行とステップ実行の動作については，次の点に注意してください。

- 1 行のソース行は，使用する言語とコーディング方法により， 1 つまたは複数のプログラミング言語要素で構成される。
- 「Step」ボタンをクリックすると，デバッガは実行可能な行を 1 行実行し，その次に実行可能な行の始めで実行を中断する。この場合，途中の実行不可能な行はスキップされる。
- 実行可能な行は，コンパイラによって生成された命令の行である (たとえば，ルーチン呼び出し文の行や代入文の行)。メイン・ウィンドウでは，実行可能な各行の左にボタンが表示される。
- 実行不可能な行には，たとえばコメント行や値を代入しない変数宣言の行がある。メイン・ウィンドウでは，実行不可能な行の左にはボタンは表示されない。

コンパイル時にコードを最適化した場合，表示されるソース・コードと実際に実行しているコードとが対応しないことがあるので注意してください (第 1.2 節を参照)。

10.3.4 呼び出されるルーチン内の命令のステップ実行

ルーチン呼び出し文でプログラムの実行が一時停止したときは，「Step」ボタンをクリックすれば，通常，呼び出されるルーチン内の命令が 1 ステップ実行されます (そのときのコーディング方法によって異なる)。そしてデバッガは，呼び出されたルーチン内にはブレークポイントが設定されていないとみなし，実行を呼び出し元ルーチン内のその次のソース行で中断します。その結果，呼び出されるルーチン (そのうちのいくつかはシステム・ルーチンかライブラリ・ルーチンである) を最後までトレース

で実行する必要がないので，コードを迅速にステップ実行できます。このことを，呼び出されるルーチンを 1 ステップとして実行するといいます。

呼び出されるルーチン内の命令をステップ実行によって 1 行ずつ実行するには，次の手順に従ってください。

1. ルーチン呼び出し文で実行を中断する。そのためには，たとえばブレークポイント (第 10.4 節を参照) を設定し，プッシュ・ボタン・ビューの「Go」ボタンをクリックする。
2. 呼び出し文で実行が一時停止したら，プッシュ・ボタン・ビューの「S/in」ボタンをクリックするか， DBG>プロンプトで STEP/INTO と入力する。その結果，実行は呼び出されたルーチンの先頭を通過する。

呼び出されたルーチン内の命令が実行され始めたら，「Step」ボタンを使用してルーチンを 1 行ずつ実行します。

ルーチン呼び出し文で実行が一時停止していないときに「S/in」ボタンをクリックすると，「Step」ボタンのクリックと同じ働きをします。

10.3.5 呼び出されたルーチンからの戻り

呼び出されたルーチン内で実行が中断しているときは，プッシュ・ボタン・ビューの「S/ret」ボタンをクリックするか， DBG>プロンプトで STEP/RETURN コマンドを入力することにより，そのルーチンの最後まで直接実行できます。

デバッガは，そのルーチンの戻り命令実行の直前で中断します。その時点で，そのルーチンの呼び出しフレームは呼び出しスタックから削除されていないので，そのルーチンにローカルな変数の値を参照することなどができます。また次の Return 命令または Call 命令までプログラムを直接実行するときは，プッシュ・ボタンの S/call ボタンを使用 (または DBG>プロンプトで STEP/CALL コマンドを入力) します。

「S/ret」ボタンは，システム・ルーチンやライブラリ・ルーチン内の命令を誤ってステップ実行した場合に特に役立ちます (第 10.1.3 項を参照)。

10.4 ブレークポイントの設定による実行の中断

ブレークポイントとは，変数の値のチェックやルーチン内の命令のステップ実行などを行うために実行をやめる必要があるプログラム内の記憶位置のことです。

デバッガの DECwindows Motif for OpenVMS ユーザ・インタフェースでは，次の各箇所にブレークポイントを設定できます。

- 指定のソース行
- 指定のルーチン (関数，サブプログラムなど)

- プログラムの実行中にシグナル通知される例外

注意

PointerGrab または KeyboardGrab によって，マウス・ポインタを制御しているルーチン内のブレークポイントで停止すると，ワークステーションはハングします。

この問題を回避するには，2 つのワークステーションを使用してプログラムをデバッグします。詳細については，第 9.8.3.1 項を参照してください。

デバッガでは次の 2 種類のブレークポイントを設定できます。

- **条件付きブレークポイント**

  指定した関係式の評価が真のときにだけ検出される。

- **アクション・ブレークポイント**

  このブレークポイントが検出されると，指定したシステム固有コマンドが 1 つまたは複数実行される。

条件付きブレークポイントであり同時にアクション・ブレークポイントでもあるブレークポイントを設定できます。

次の各項ではこれらのブレークポイント・オプションについて説明します。

10.4.1 ソース行へのブレークポイントの設定

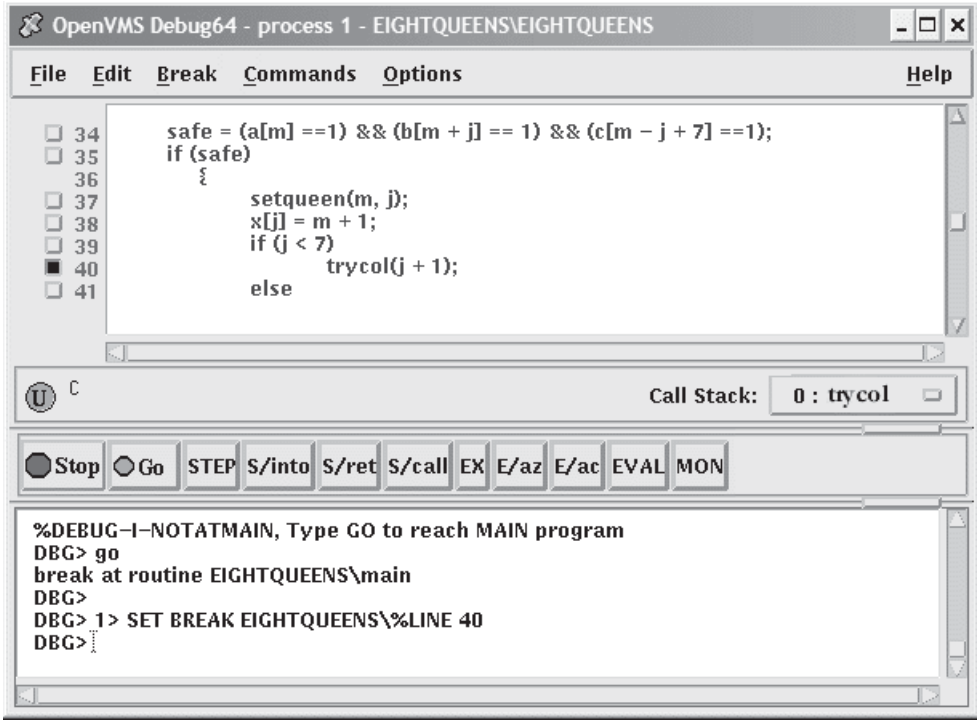
ソース・ディスプレイのソース行のうち，その左にボタンがあるソース行には，ブレークポイントを設定できます。ボタンが表示されている行は，コンパイラが実行可能コードを生成した行 (ルーチン宣言，代入文など) です。

ソース行にブレークポイントを設定するには，次の手順に従ってください。

1. ブレークポイントを設定するソース行を検索する (第 10.1 節を参照)。
2. その行の左にあるボタンをクリックして選択する。ボタンが選択されるとブレークポイントが設定される。ブレークポイントは，ソース行の先頭，つまりそのソース行に対応する最初の機械語コード命令に設定される。

図 10–4 では， 37 行目の先頭にブレークポイントが設定されている。

図 10–4 ソース行へのブレークポイントの設定



10.4.2 ソース・ブラウザによるルーチン上のブレークポイントの設定

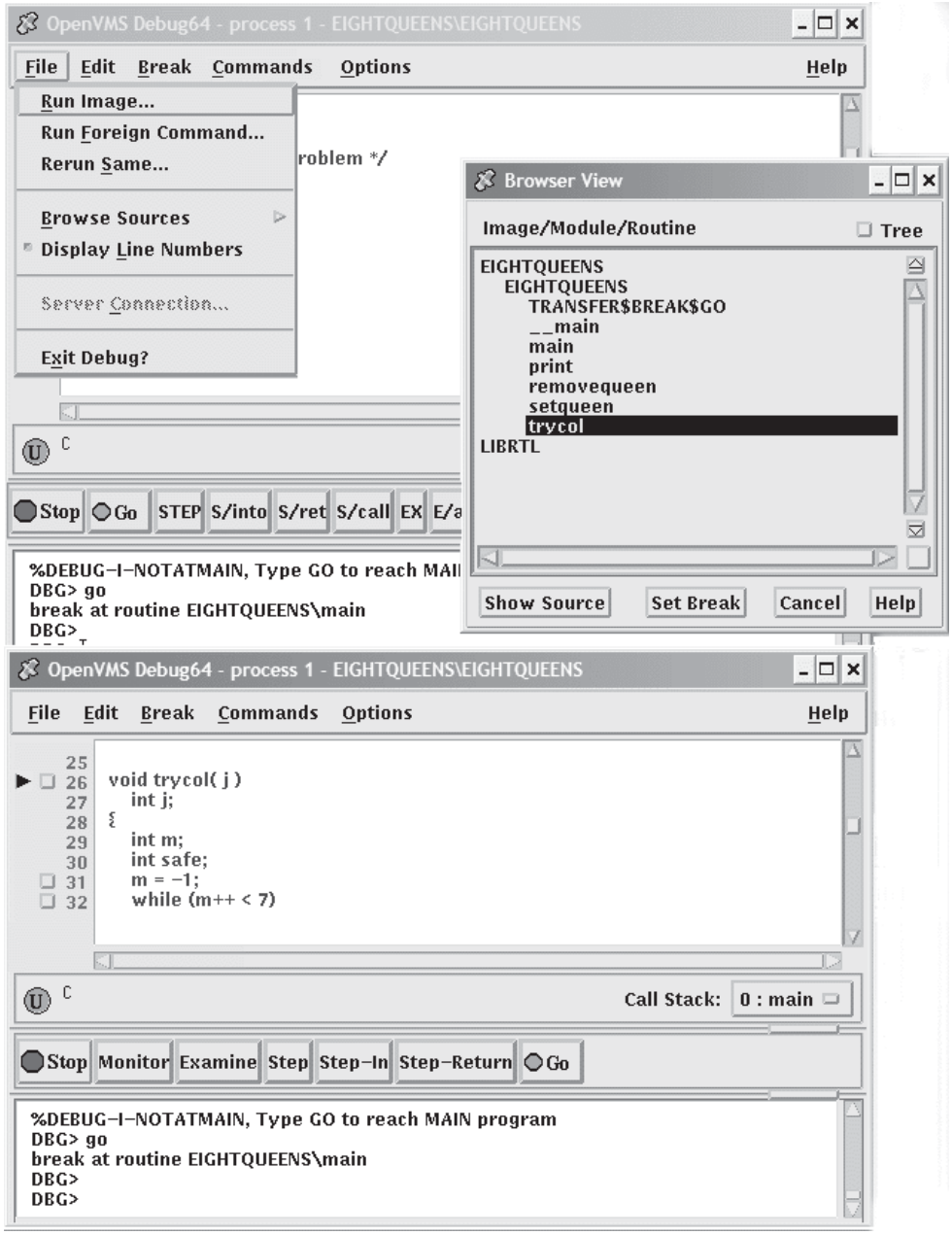
ルーチンにブレークポイントを設定すれば，そのルーチンまで直接実行を進めてそのローカル環境を検査することができます。

ルーチンにブレークポイントを設定するには，次の手順に従ってください。

1. メイン・ウィンドウの「File」メニューで「Browse Sources...」を選択する (図 10–2 を参照)。「Source Browser」ダイアログ・ボックスに，実行可能なイメージおよびこれとリンクされたすべての共用可能イメージ (たとえば， DEBUG および LIBRTL) の名前が表示される。実行可能なイメージは強調表示される。このイメージにシンボリック情報がない場合，リンクされたイメージの名前が薄く表示される。
2. 実行可能なイメージの名前をダブル・クリックする。そのイメージ名の下にインデントされて，そのイメージ内の各モジュールの名前が表示される。
3. 表示したいモジュールの名前をダブル・クリックする。モジュール名の下に，そのモジュール内のルーチンの名前が (インデント付きで) 表示される (図 10–5 を参照)。
4. ブレークポイントを設定するルーチンの名前をクリックする。 Set Breakpoint コマンドの結果が，コマンド・ビューのコマンド行にエコーバックされる。

また別の方法として，ルーチン名をクリックした後，「Source Browser」ビューの「Set Breakpoint」ボタンをクリックする方法もある。この場合も， Set Breakpoint コマンドの結果が，コマンド・ビューのコマンド行にエコーバックされる。

図 10–5 ルーチンへのブレークポイントの設定



10.4.3 例外ブレークポイントの設定

例外ブレークポイントを設定すると，例外がシグナル通知されたとき，ユーザ・プログラムによって宣言された例外ハンドラが実行される前に実行が中断されます。したがって例外ハンドラが使用できる場合は，その中の命令をステップ実行することにより，制御の流れをチェックできます。

例外ブレークポイントを設定するには，メイン・ウィンドウまたはオプション・ビュー・ウィンドウの「Break」メニューから「On Exception」を選択します。例外がシグナル通知されるたびに例外ブレークポイントが検出されます。

10.4.4 現在設定されているブレークポイントの識別

現在設定されているブレークポイントを次の 3 つの方法で知ることができます。

- ブレークポイント・ボタンが塗りつぶされている行に注意しながら，ソース・コードをスクロールする。この方法では時間がかかり，しかも一度設定されたあとに無効にされたブレークポイントは表示されない (第 10.4.5 項を参照)。
- メイン・ウィンドウまたはオプション・ビュー・ウィンドウの「Options」メニューで「Views...」を選択する。「Views」ダイアログ・ボックスが表示されたら，「Breakpoint View」をクリックしてブレークポイント・ビューを表示する (図 8–4 を参照)。

  ブレークポイント・ビューには，各ブレークポイントのモジュール名と行番号の一覧が表示される (第 10.1 節を参照)。表示された各ブレークポイントの隣にある塗りつぶされたボタンは，そのブレークポイントが有効なことを示す。塗りつぶされていないボタンは，そのブレークポイントが無効なことを示す。
- コマンド・ビューの DGB>プロンプトで SHOW BREAK コマンドを入力する。デバッガには，現在設定されているすべてのブレークポイントがリストされ，条件ブレークポイントの場合のトリガ条件やアクション・ブレークポイントの場合に実行されるコマンドもあわせて表示される。

10.4.5 ブレークポイントの無効化，有効化，取り消し

ブレークポイントを設定すると，無効化，有効化または削除を行うことができます。

無効にしたブレークポイントは，プログラムの実行中はデバッガによって無視されます。しかし，そのブレークポイントはブレークポイント・ビューに表示されているので，あとで，たとえばプログラムの再実行時などに有効にできます (第 9.3 節を参照)。次の点に注意してください。

- 特定のブレークポイントを無効にするには，メイン・ウィンドウ内かブレークポイント・ビュー内のそのブレークポイントのボタンをクリアする。

すべてのブレークポイントを無効にするには，「Break」メニューで「Deactivate All Breaks」を選択する。

- ブレークポイント・ビューの場合，ブレークポイントが現在有効であれば，「Break」メニューの「Toggle」を選択することもできる。

あるブレークポイントを有効にすれば，そのブレークポイントはプログラムの実行中有効になります。

- あるブレークポイントを有効にするには，メイン・ウィンドウ内かブレークポイント・ビュー内のそのブレークポイントのボタンを選択して塗りつぶす。

  ブレークポイント・ビューの場合，ブレークポイントが現在無効であれば，「Break」メニューの「Toggle」を選択することもできる。
- すべてのブレークポイントを有効にするには，「Break」メニューで「Activate All」を選択する。

あるブレークポイントを取り消すと，そのブレークポイントはブレークポイント・ビューに表示されなくなり，あとでビューを使用して有効にできなくなります。第 10.4.1 項と第 10.4.2 項の説明に従ってブレークポイントを再設定する必要があります。次の点に注意してください。

- 特定のブレークポイントを削除するには，オプション・ビュー・ウィンドウの「Break」メニューで「Cancel」を選択する。
- すべてのブレークポイントを削除するには，「Break」メニューの中から「Cancel All」を選択する。

10.4.6 条件付きブレークポイントの設定

条件付きブレークポイントで実行が中断されるのは，指定された式の評価が真のときだけです。たとえば，プログラム内のある変数の値が 4 のときにブレークポイントが有効になるように指定することができます。その変数の値が 4 でなければ，そのブレークポイントは無視されます。

プログラムの実行中にブレークポイントが検出されると，条件式が評価されます。

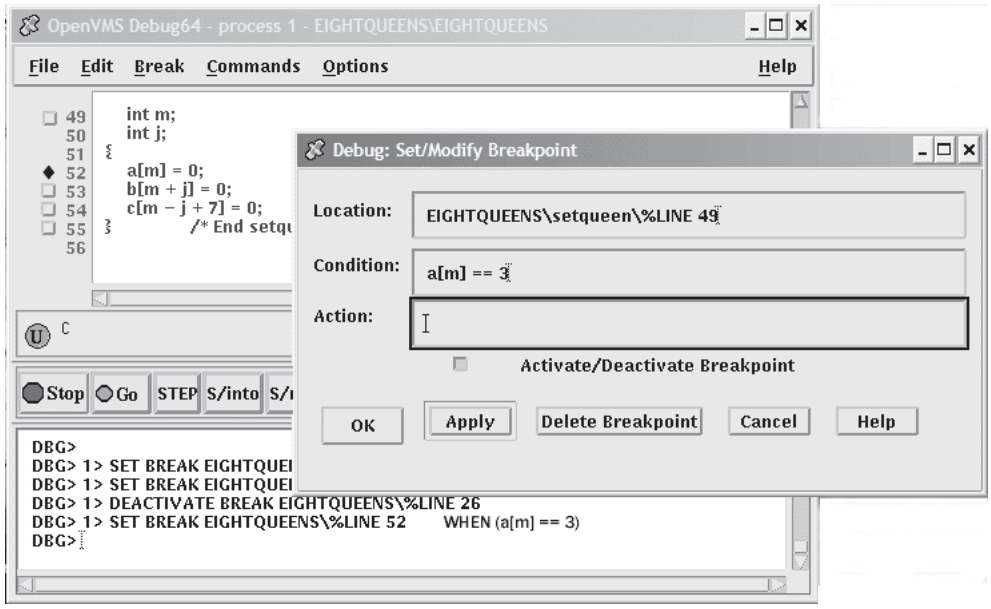
次の手順により，条件付きブレークポイントが設定されます。その場合，以前同じ場所にブレークポイントが設定されていたかどうかは関係ありません。

1. 条件付きブレークポイントの設定先のソース行を表示する (第 10.1 節を参照)。
2. 次のいずれかを実行する。
   - ソース行の左にあるボタン上で Ctrl/MB1 を押す。「Set/Modify Breakpoint」ダイアログ・ボックスが表示され，選択されているソース行が「Location:」フィールドに表示される (図 10–6 を参照)。

- 「Break」メニューから「Set」または「Set/Modify」を選択する。「Set /Modify Breakpoint」ダイアログ・ボックスが表示されたら，「Location:」フィールドにソース行を入力する。

3. ダイアログ・ボックスの「Condition:」フィールドに関係式を入力する。その関係式は，ソース言語で有効なものでなければならない。たとえば，`a[3] == 0`は C 言語で有効な関係式である。
4. 「OK」をクリックする。条件付きブレークポイントが設定される。ブレークポイントのボタンの形が四角からダイヤに変わり，ブレークポイントが条件付きであることが示される。

図 10–6 条件付きブレークポイントの設定



条件付きブレークポイントを変更するには，次の手順に従ってください。この手順で，既存の条件付きブレークポイントの位置や割り当てた条件を変更したり，無条件ブレークポイントを条件付きブレークポイントに変更することができます。

1. メイン・ウィンドウまたはオプション・ビュー・ウィンドウの「Options」メニューで「Views...」を選択する。「Views」ダイアログ・ボックスが表示されたら，「Breakpoint View」をクリックしてブレークポイント・ビューを表示する。
2. ブレークポイント・ビューで次のいずれかを実行する。
   - 表示されているブレークポイントの左にあるボタン上で Ctrl/MB1 を押す。
   - ビューに表示されているブレークポイントをクリックしてから「Break」メニューで「Set/Modify」を選択する。

3. 前の手順の 3 と 4 を実行し，適切な設定を行う。

10.4.7 アクション・ブレークポイントの設定

アクション・ブレークポイントが検出されると，実行は中断され，指定したコマンドの並びが実行されます。

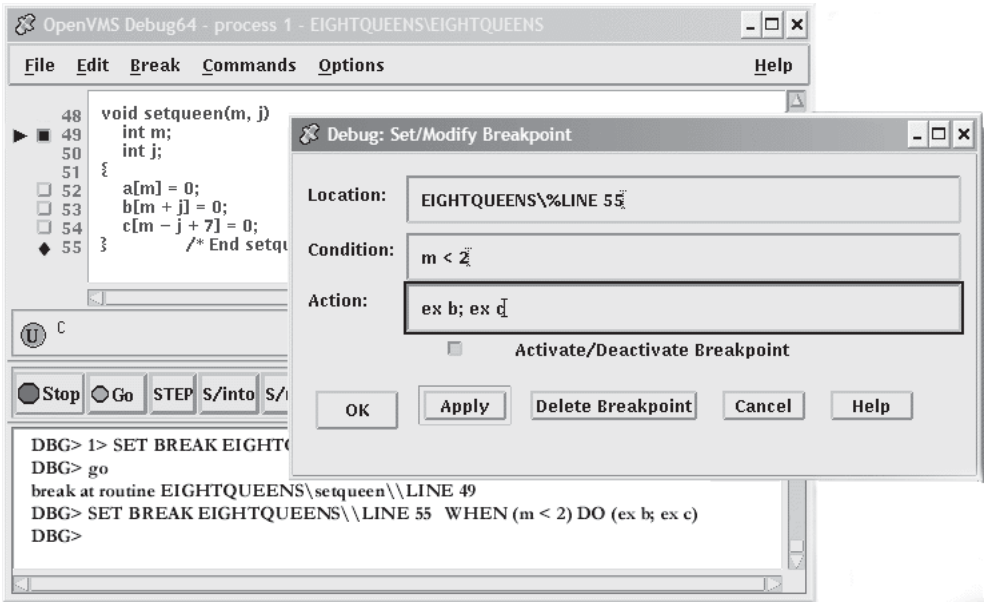
アクション・ブレークポイントを設定するには，次の手順に従ってください。以前に同じ場所にブレークポイントが設定されていたかどうかは関係ありません。

1. アクション・ブレークポイントの設定先のソース行を表示する (第 10.1 節を参照)。
2. 次のいずれかを実行する。
   - ソース行の左にあるボタン上で Ctrl/MB1 を押す。「Set/Modify Breakpoint」ダイアログ・ボックスが表示され，選択されているソース行が「Location:」フィールドに表示される (図 10–6 を参照)。
   - 「Break」メニューで「Set」または「Set/Modify」を選択する。「Set /Modify Breakpoint」ダイアログ・ボックスが表示されたら，「Location:」フィールドにソース行を入力する。
3. ダイアログ・ボックスの「Action:」フィールドに 1 つまたは複数のデバッガ・コマンドを入力する。たとえば，`DEPOSIT x[j] = 3; STEP; EXAMINE a`と入力する。
4. 「OK」をクリックする。アクション・ブレークポイントが設定される (図 10–7 を参照)。

アクション・ブレークポイントを変更するには，次の手順に従ってください。この手順で，既存のアクション・ブレークポイントの位置や割り当てた条件を変更したり，無条件ブレークポイントをアクション・ブレークポイントに変更することができます。

1. メイン・ウィンドウまたはオプション・ビュー・ウィンドウの「Options」メニューで「Views...」を選択し，「Views」ダイアログ・ボックスが表示されたら，「Breakpoint View」をクリックする。
2. ブレークポイント・ビューで次のいずれかを実行する。
   - 表示されているブレークポイントの左にあるボタン上で Ctrl/MB1 を押す。
   - ビューに表示されているブレークポイントをクリックしてから「Break」メニューで「Set/Modify」を選択する。
3. 前の手順の 3 と 4 を実行し，適切な設定を行う。

図 10–7 アクション・ブレークポイントの設定



10.5 変数の検査と操作

この節では，次の操作方法について説明します。

- ウィンドウでの変数名の選択
- 変数の値の表示
- 変数のモニタ
- 変数のウォッチ
- 変数の値の変更

変数の操作全般については，第 10.6 節も参照してください。

10.5.1 ウィンドウでの変数名の選択

次の各項の操作では，次の方法でウィンドウから変数を選択します。例を第 10.5.2 項に示します。

名前を選択するときは，ソース・プログラミング言語の構文に従います。

- スカラ (非集合体) 変数，たとえば整数型，実数型，論理型，または列挙型の変数を指定するには，その変数の名前を選択する。
- 集合体全体，たとえば配列や構造体 (レコード) を指定するには，その変数の名前を選択する。

- 集合体変数の要素を指定するには，その言語の構文を使用してその要素を選択する。たとえば，次のとおりです。
  - 文字列arr2[7]は， C 言語の配列arr2の要素 7 を指定する。
  - 文字列employee.addressは， Pascal 言語のレコード (構造体) employeeの構成要素addressを指定する。
- ポインタ変数が示すオブジェクトを指定するには，その言語の構文に従ってその要素を選択する。たとえば C 言語では，文字列*int_pointは，ポインタint_pointによって示されるオブジェクトを指定する。

ウィンドウ内の文字列は次のようにして選択します。

- どのウィンドウの場合も，ブランクで区切られている文字列を選択するには，DECwindows Motif for OpenVMS の標準の単語選択方法 (目的の文字列にポインタを位置づけて MB1 をダブル・クリックする) を使用する。
- どのウィンドウの場合も，任意の文字列を選択するには， DECwindows Motif for OpenVMS の標準のテキスト選択方法 (1 文字目にポインタを位置づけ，MB1 を押したままそのポインタを文字列の最後までドラッグし， MB1 を離す) を使用する。
- デバッガのソース・ディスプレイの場合，言語依存のテキスト選択を行うこともできる。言語依存識別子の境界で区切られた文字列を選択するには，ポインタをその文字列に位置づけ， Ctrl/MB1 を押す。

  たとえば，ソース表示に文字列arr2[m]が含まれている場合は，次のようにする。
  - arr2を選択するには，ポインタをarr2に位置づけ， Ctrl/MB1 を押す。
  - mを選択するには，ポインタをmに位置づけ， Ctrl/MB1 を押す。

  言語依存のテキスト選択のキー順序は，第 10.10.4.2 項の説明に従って変更できる。

10.5.2 変数の現在値の表示

変数の現在値を表示するには，次の手順に従います。

1. 第 10.5.1 項の説明に従って，ウィンドウ内の変数名を検索し，選択する。
2. プッシュ・ボタン・ビューの「EX」ボタンをクリックする。コマンド・ビューに変数とその現在値が表示される。この値は現在の有効範囲内の変数の値であり，ユーザが変数を選択したソース記憶位置の値ではない。

図 10–8，図 10–9，および図 10–10 に整変数，配列集合体，および配列要素の表示方法をそれぞれ示します。

図 10–8 整変数の値の表示

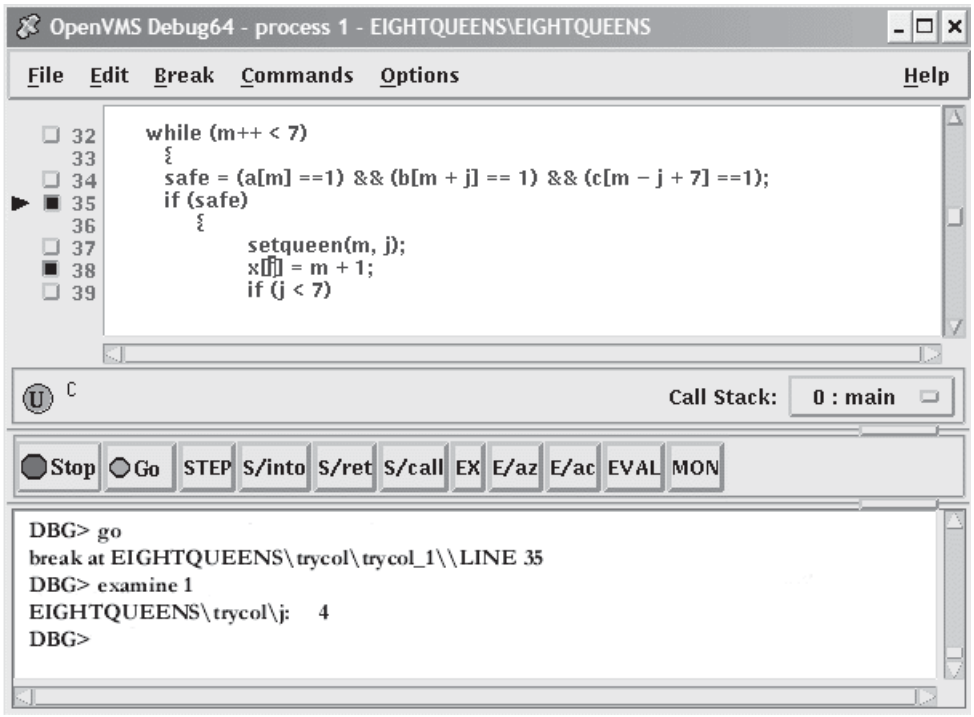


図 10–9 配列集合体の値の表示

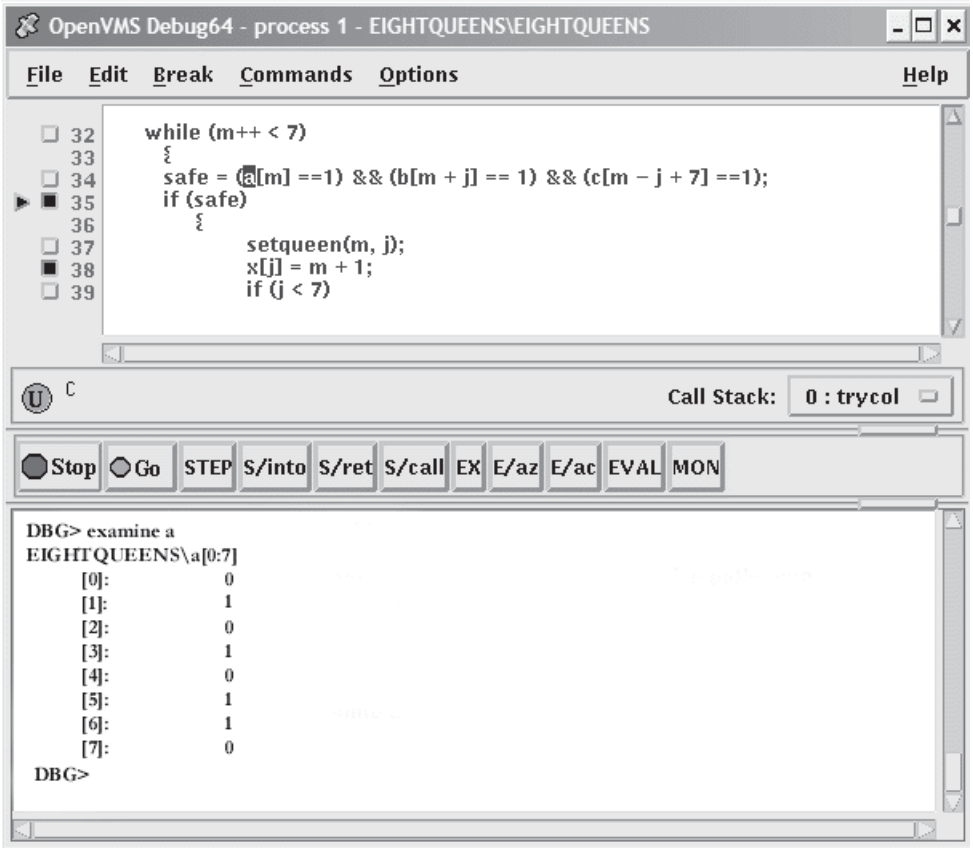
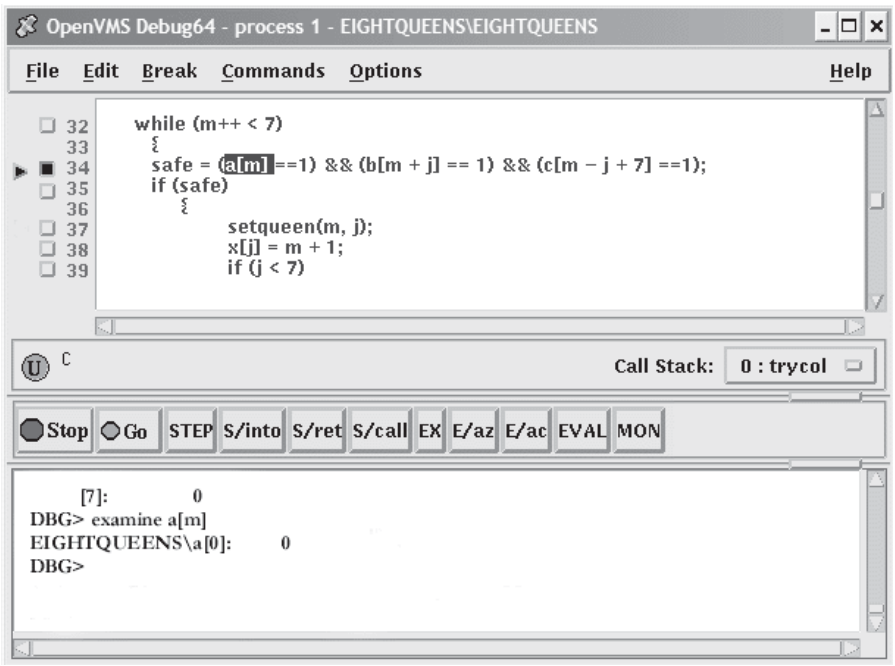


図 10–10 配列集合体の値の表示



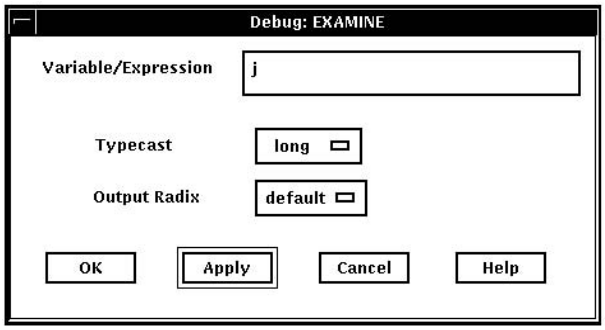
現在値を別の型や基数で表示するには，次の手順に従ってください。

1. 第 10.5.1 項の説明に従って，ウィンドウ内で変数名を検索して選択する。
2. メイン・ウィンドウの「Command」メニューで「Examine...」を選択する。「Examine」ダイアログ・ボックスが表示され，選択されている変数名が「Variable/Expression」フィールドに表示される。
3. ダイアログ・ボックス内の「Typecast」メニューで「default」，「int」，「long」，「quad」，「short」，または「char*」のいずれかを選択する。
4. ダイアログ・ボックス内の「Output Radix」メニューで「default」，「hex」，「octal」，「decimal」，または「binary」のいずれかを選択する。
5. 「OK」をクリックする。

指定に応じて変更された値がコマンド・ビューに表示されます。

図 10–11 では，変数 j が long に型キャストされています。

図 10–11 変数値の型キャスト



10.5.3 変数の現在値の変更

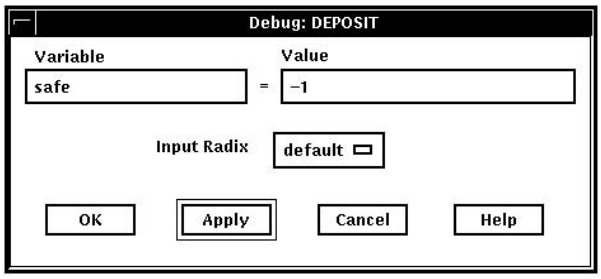
変数の現在値を変更するには，次の手順に従ってください。

- 第 10.5.1 項の説明に従って，ウィンドウ内で変数名を検索して選択する。
- メイン・ウィンドウの「Command」メニューで「Deposit...」を選択する。「Deposit」ダイアログ・ボックスが表示され，選択されている変数名が「Variable」フィールドに表示される。
- 「Value」フィールドに新しい値を入力する。
- ダイアログ・ボックス内の「Input Radix」メニューで「default」，「hex」，「octal」，「decimal」，「binary」のいずれかを選択する。
- 「OK」をクリックする。

指定に応じた新しい値がコマンド・ビューに表示され，変数に代入されます。

図 10–12 では，変数 safe の値を変更しています。

図 10–12 変数値の変更



10.5.4 変数のモニタ

変数をモニタする場合，デバッガはその値をモニタ・ビューに表示します。また，たとえば，ステップのあとやブレークポイントでプログラムからデバッガに制御が戻ると，表示されている値をチェックし，更新します。

注意

モニタできるのは，変数と，配列や構造体 (レコード) などの集合体だけです。複合式やメモリ・アドレスはモニタできません。

変数をモニタするには，次の手順に従ってください (図 10–13 を参照)。

1. 第 10.5.1 項の説明に従って，ウィンドウ内で変数名を検索し，選択する。
2. プッシュ・ボタン・ビューの「MON」ボタンをクリックする。デバッガは次の表示を行う。
   - モニタ・ビューを表示する (まだ表示されていない場合)。
   - 選択された変数の名前とその修飾パス名を「Monitor Expression」欄に表示する。
   - その変数の値を「Value/Deposit」欄に表示する。
   - 塗りつぶされていないボタンを「Watched」欄に表示する (第 10.5.5 項を参照)。

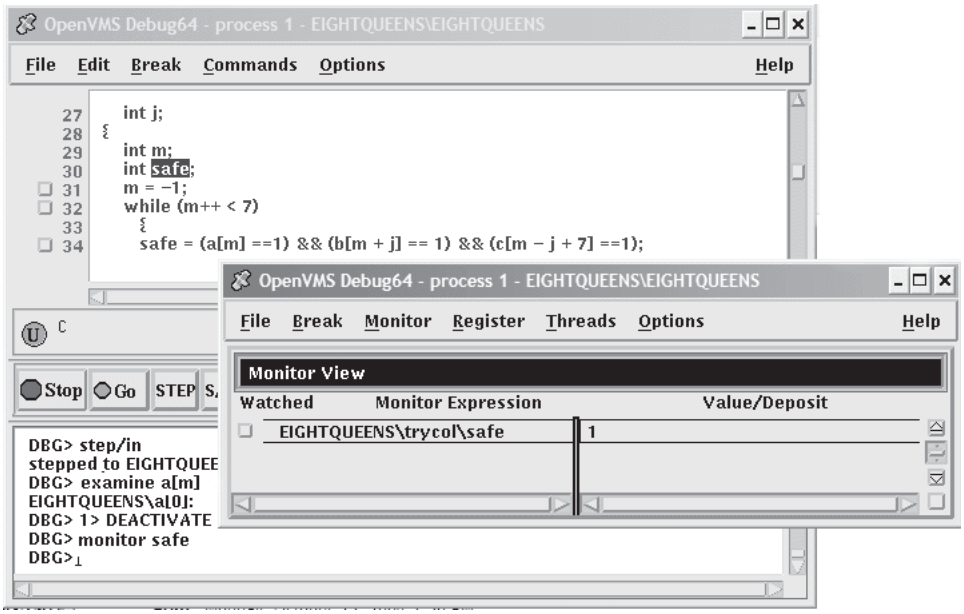
変数をモニタしているときに出力値を型キャストするには，「Monitor」メニューで「Typecast」を選択します。

モニタ中の変数の出力の基数は次の方法で変更できます。

- モニタ中の選択された要素の出力の基数を変更するには，「Monitor」メニューで「Change Radix」を選択する。
- これ以降モニタするすべての要素の出力の基数を変更するには，「Monitor」メニューで「Change All Radix」を選択する。

モニタしている要素をモニタ・ビューから削除するには，「Monitor」メニューで「Remove」を選択します。

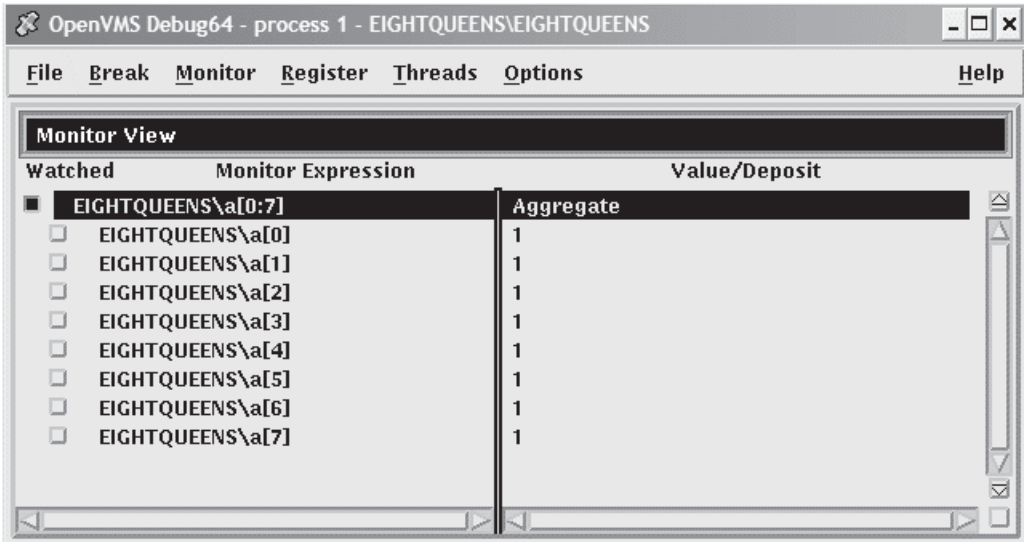
図 10–13 変数のモニタ



10.5.4.1 集合体 (配列または構造体) 変数のモニタ

配列や構造体 (レコード) などの集合体変数の名前を選択し，「MON」ボタンをクリックすると，モニタ・ビューの「Value/Deposit」欄にAggregateという語が表示されます。集合体変数のすべての要素 (構成要素) の値を表示するには，「Monitor Expression」欄の変数名をダブル・クリックするか，または「Monitor」メニューで「Expand」を選択します。各要素の名前は，親の名前よりインデントされて表示されます (図 10–14 を参照)。ある要素が集合体の場合，その名前をダブル・クリックすればさらにその要素も表示されます。

図 10–14　モニタ・ビューに展開された集合体変数 (配列)



拡大された表示を元に戻して，集合体の親の名前だけをモニタ・ビューに表示するには，「Monitor Expression」欄の変数名をダブル・クリックするか，または「Monitor」メニューで「Collaspe」を選択します。

集合体変数の構成要素を選択した場合，その構成要素式自身が変数であれば，選択時にアクティブだった構成要素がモニタされます。たとえば，配列構成要素arr[i]を選択した場合，iの現在値が 9 であれば，たとえiの値があとで 10 に変わっても，arr[9]がモニタされます。

10.5.4.2　ポインタ (アクセス) 変数のモニタ

ポインタ (アクセス) 変数の名前を選択し「MON」ボタンをクリックすると，参照されたオブジェクトのアドレスがモニタ・ビューの「Value/Deposit」欄に表示されます (図 10–15 の最初のエントリを参照)。

参照されたオブジェクト値をモニタする (ポインタ変数を間接参照する) には，「Monitor Expression」欄のポインタ名をダブル・クリックします。この結果，モニタ・ビューではそのポインタ変数のエントリの下に，参照されたオブジェクトエントリがインデントされて表示されます (図 10–15 の一番下のエントリを参照)。参照されたオブジェクトが集合体の場合，その名前をダブル・クリックすればさらにその要素も表示されます。

図 10–15 モニタ・ビューでのポインタ変数と参照されたオブジェクト

10.5.5 変数のウォッチ

ウォッチされている変数の値がプログラムで変更されると，実行が中断しコマンド・ビューにその新旧の値が表示されます。

変数をウォッチする (変数へのウォッチポイントの設定) には，次の手順に従ってください。

- 第 10.5.4 項の説明に従って変数をモニタする。変数をモニタするたびに，モニタ・ビューの「Watched」欄にボタンが 1 つ表示される (図 10–16 を参照)。
- 「Watched」欄のボタンをクリックする。ボタンが塗りつぶされて，ウォッチポイントの設定が示される。

図 10–16 モニタ・ビューでの変数のウォッチ

ウォッチポイントを無効にするには，モニタ・ビューの「Watched」ボタンをクリックしてクリアするか，または「Monitor」メニューで「Toggle Watchpoint」を選択します。ウォッチポイントを有効にするには，目的の「Watched」ボタンをクリックして塗りつぶすか，または「Monitor」メニューで「Toggle Watchpoint」を選択します。

静的変数と非静的 (自動) 変数，およびこれらの変数へのアクセス方法については，第 10.6.1 項を参照してください。変数の定義元ルーチンから実行の制御が移る (戻る) と，非静的ウォッチポイントは無効になります。非静的変数がアクティブでなくなると，モニタ・ビューではそのエントリが薄く表示され，その「Watched」ボタンはクリアされます。

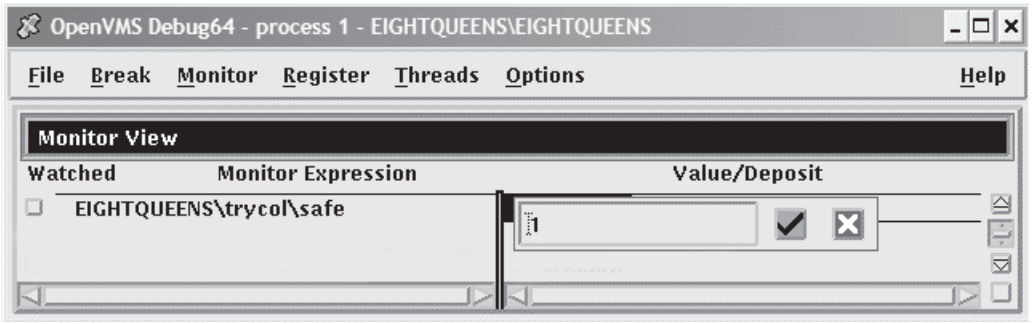
変数の定義元ルーチンに実行の制御が後で戻る場合，非静的ウォッチポイントが自動的に再び有効になることはありません。非静的ウォッチポイントは，ユーザが自分で明示的に，有効にしなければなりません。

10.5.6 モニタされたスカラ型変数の値の変更

スカラ (非集合体) 型変数，たとえば整数型や論理型の値を変更するには，次の手順に従ってください (図 10–17 を参照)。

1. 第 10.5.4 項の説明に従って変数をモニタする。
2. モニタ・ビューの「Value/Deposit」欄の変数値をクリックする。その値の上に小さな編集可能ダイアログ・ボックスが表示される。
3. そのダイアログ・ボックスに新しい値を入力する。
4. そのダイアログ・ボックスのチェック・マーク (OK) をクリックする。ダイアログ・ボックスが消えて新しい値が表示され，変数がその値になったことが示される。その変数の型，範囲などに合わない値を入力しようとすると注意が表示される。

図 10–17 モニタされたスカラ型変数の値の変更

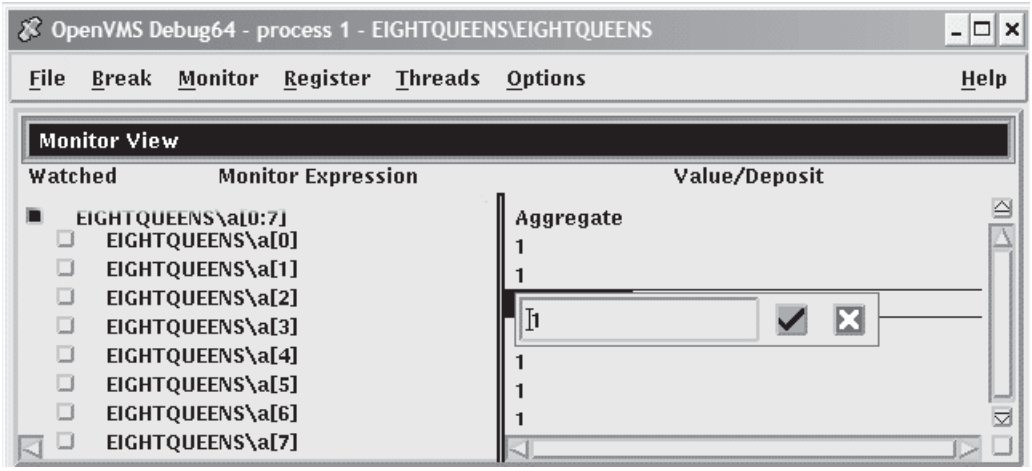


値の入力を中止してダイアログ・ボックスを消すには， X (取り消し) をクリックします。

集合体型変数 (たとえば配列や構造体) は一度に 1 つの構成要素の値を変更できます。集合体型変数の構成要素の値を変更するには，次の手順に従ってください (図 10–18 を参照)。

1. 第 10.5.4.1 項の説明に従って構成要素の値を表示する。
2. スカラ型変数の値の変更手順に従う。

図 10–18 集合体型変数の構成要素の値の変更



10.6 プログラム変数へのアクセス

この節では，デバッグ時にプログラム変数にアクセスするときの一般的な関連事項について説明します。

コンパイル時にプログラムを最適化すると，デバッグ時に特定の変数にアクセスできなくなります。デバッグ対象のプログラムをコンパイルするときは，できるだけ最適化しないようにしてください (第 1.2 節を参照)。

変数の値をチェックする前に，その変数が宣言され初期化される箇所の先まで必ずそのプログラムを実行します。初期化されていない変数内の値は，不当であると考えられます。

10.6.1 静的変数と非静的 (自動) 変数へのアクセス

注意

ここでは総称して「非静的変数」という用語を使用しますが，言語によっては自動変数と呼ぶ場合もあります。

静的変数には，プログラムの実行中は同じメモリ・アドレスが割り当てられます。静的変数にはいつでもアクセスできます。

非静的変数はスタックかレジスタに割り当てられ，その定義元ルーチンかブロックが呼び出しスタック上でアクティブな場合にだけ値を持ちます。したがって，非静的変数にアクセスできるのは，その定義元ルーチンかブロック (定義元ルーチンから呼び

出されたルーチンを含む) の有効範囲内でプログラムの実行が一時停止しているときだけです。

通常，非静的変数にアクセスするには，定義元にまずブレークポイントを設定してから，そのブレークポイントまでプログラムを実行します。

ユーザ・プログラムの実行によって非静的変数がアクセスできなくなると，次のような通知がなされます。

- 変数の値を表示または変数をモニタしようとしている場合は，それぞれ第 10.5.2 項と第 10.5.4 項の説明のように，その変数がアクティブでないまたは有効範囲にないというメッセージが発行される。
- 変数 (または変数を含む式) をモニタしている場合は，モニタ・ビュー内のそのエントリが薄く表示される。エントリが薄く表示されていると，その変数の表示値はチェックも更新もされない。また，第 10.5.3 項のように，その値を変更することもできない。そのエントリは，その変数が再びアクセス可能になれば普通に表示される。
- 変数をウォッチしている場合は (第 10.5.5 項を参照)，そのウォッチポイントは無効になり (そのウォッチ・ボタンはクリアされる)，そのエントリは薄く表示される。ただし，その変数が再びアクセス可能になっても，そのウォッチポイントは自動的には再度有効にならないことに注意。

10.6.2 呼び出しスタックを基準とする現在の有効範囲の設定

プログラム内のルーチンをデバッグしているときに，現在の有効範囲を呼び出し元ルーチン (スタックで，現在実行が一時停止しているルーチンより下にあるルーチン) に設定することができます。これにより，次のことが可能になります。

- 現在のルーチン呼び出しの開始位置を明らかにする。
- 呼び出し元ルーチンで宣言されている変数の値を明らかにする。
- 再帰呼び出しされるルーチンの特定の起動時に変数の値を明らかにする。
- ルーチン呼び出しの場合の変数の値を変更する。

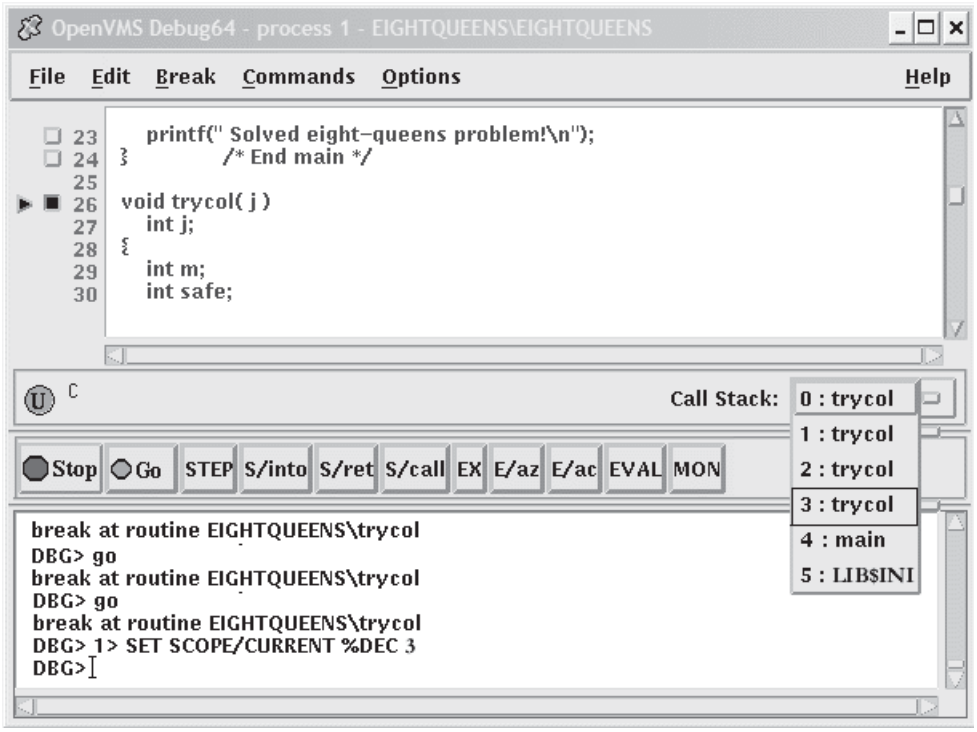
メイン・ウィンドウの「Call Stack」メニューには，スタックで現在アクティブなプログラム・ルーチン (および，特定の条件下でのイメージとモジュール) の名前が，画面に表示できる最大行数まで一覧表示されます (図 10–19 を参照)。メニューの左側の番号は，実行が一時停止しているルーチンをレベル 0 としたときのスタックの各ルーチンのレベルです。

現在の有効範囲をスタックの特定のルーチンに設定するには，「Call Stack」メニュー (図 10–19 を参照)。からその名前を選択します。すると，次の処理が行われます。

- 「Call Stack」メニューには，現在の有効範囲であるルーチンの名前と相対レベルが表示される。

- メイン・ウィンドウには，そのルーチンのソース・コードが表示される。
- 命令ビューには，そのルーチンのデコード済み命令が表示される。
- レジスタ・ビューには，そのルーチン呼び出しに対応したレジスタ値が表示される。
- 有効範囲を呼び出し元ルーチン (0 以外の呼び出しスタック・レベル) に設定すると，現在位置ポインタは図 10–19 のように白抜きになる。
- シンボル検索の有効範囲は選択されたルーチンに設定されるので，その有効範囲内で変数を検査することなどができる。

図 10–19 現在の有効範囲を呼び出し元ルーチンに設定する



有効範囲をある呼び出し元ルーチンに設定すると，その呼び出し元ルーチンに実行制御が戻ったときに実行されるソース行が，白抜きの現在位置ポインタによって示されます。ソース言語やコーディング方法により，呼び出し文を含んでいる行が示されたり，それ以降の行が示されることもあります。

10.6.3 変数やその他のシンボルの検索方法

シンボルがあいまいになるのは，シンボル (たとえば，変数名 X) を 2 つ以上のルーチンに定義するとき，つまりその他のプログラム・ユニットに定義するときです。

ほとんどの場合，シンボルのあいまいさは自動的に解消されます。まず，現在設定されている言語の有効範囲と可視性の規則が使用されます。また，デバッガではブレークポイントを設定するためなどに任意のモジュール内にシンボルを指定できるので，呼び出しスタック上のルーチン呼び出しの順序によってシンボルのあいまいさが解消されます。

しかし，複数回定義されたシンボルを指定すると，次のような処置がとられることがあります。

- ユーザの意図するシンボルの特定の宣言をデバッガが決められず，"symbol not unique"メッセージが発行される。
- 現在の有効範囲内で可視のシンボル宣言があれば，ユーザの意図するものでなくてもそれが参照される。

これらの問題を解決するためには，目的のシンボル宣言の検索範囲を指定しなければなりません。

- 呼び出しスタック内で現在アクティブな複数のルーチン内にシンボルのさまざまな宣言がある場合は，メイン・ウィンドウの「Call Stack」メニューを使用して現在の有効範囲を再設定する (第 10.6.2 項を参照)。
- それ以外の場合は，シンボルの前にパス名を指定して，適切なコマンド (EXAMINE や MONITOR など) をコマンド・プロンプトで入力する。たとえば，変数 X を COUNTER と SWAP という 2 つのモジュールに定義している場合，次のコマンドを使用すればパス名 SWAP\X によって SWAP モジュール内の X の宣言を指定できる。

```
DBG> EXAMINE SWAP\X
```

10.7 レジスタに格納されている値の表示と変更

レジスタ・ビューには，すべての機械語レジスタの現在の内容が表示されます (図 10–20 を参照)。

レジスタ・ビューを表示するには，メイン・ウィンドウまたはオプション・ビュー・ウィンドウの「Options」メニューで「Views...」を選択し，「Views」ダイアログ・ボックスが表示されたら，「Registers」をクリックします。

省略時の設定では，レジスタ・ビューには現在実行が一時停止しているルーチンに対応するレジスタ値が自動的に表示されます。プログラムからデバッガに制御が戻ると，プログラムの実行で変更されたすべての値が強調表示されます。

呼び出しスタック内の任意のルーチンに対応するレジスタ値を表示するには，メイン・ウィンドウの「Call Stack」メニューでその名前を選択します (第 10.6.2 項を参照)。

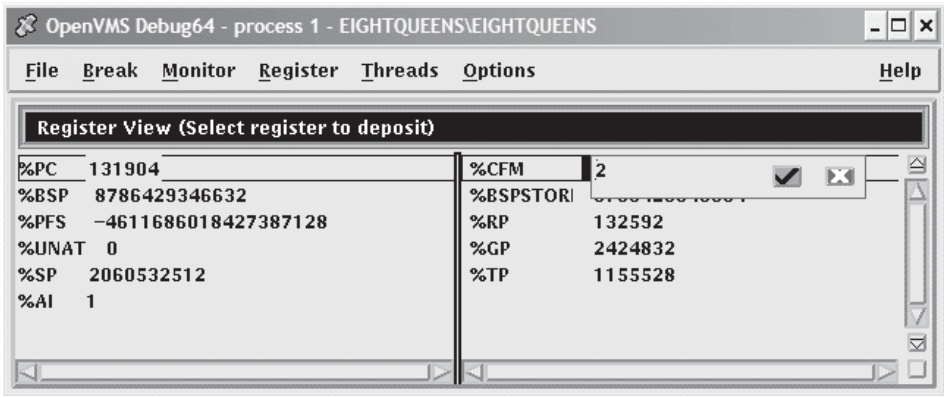
レジスタに格納されている値を変更するには，次の手順に従ってください。

1. レジスタ・ビュー内のレジスタ値をクリックする。小さな編集可能ダイアログ・ボックスが現在値の上に表示される。
2. そのダイアログ・ボックスに新しい値を入力する。
3. ダイアログ・ボックス内のチェック・マーク (OK) をクリックする。ダイアログ・ボックスが消えて新しい値が表示され，その値がレジスタに入ったことが示される。レジスタの値を変更せずにダイアログ・ボックスを消去するときは， X (Cancel) をクリックする。

レジスタ値の表示に使用する基数は，次の方法で変更できます。

- 選択されたレジスタの現在の出力とそれ以降の出力の基数を変更するには，「Register」メニューで「Change Radix」を選択する。
- すべてのレジスタの現在の出力とそれ以降の出力の基数を変更するには，「Register」メニューで「Change All Radix」を選択する。

図 10–20 レジスタ・ビュー



10.8 ユーザ・プログラムのデコード済み命令ストリームの表示

命令ビューには，ユーザ・プログラムのデコード済み命令ストリーム，つまり実際に実行されているコード (図 10–21 を参照) が表示されます。命令ビューが役立つのは，コンパイラによって最適化されたプログラムのデバッグ中は，メイン・ウィンドウの情報が実行中のコードと正確に対応していない場合です (第 1.2 節を参照)。

命令ビューを表示するには，メイン・ウィンドウまたはオプション・ビュー・ウィンドウの「Options」メニューで「Views...」を選択し，「Views」ダイアログ・ボックスが表示されたら，「Instructions」をクリックします。

省略時の設定では，命令ビューには現在実行が一時停止しているルーチンのデコード済み命令ストリームが自動的に表示されます。命令の左にある現在位置ポインタは，次の実行命令を示します。

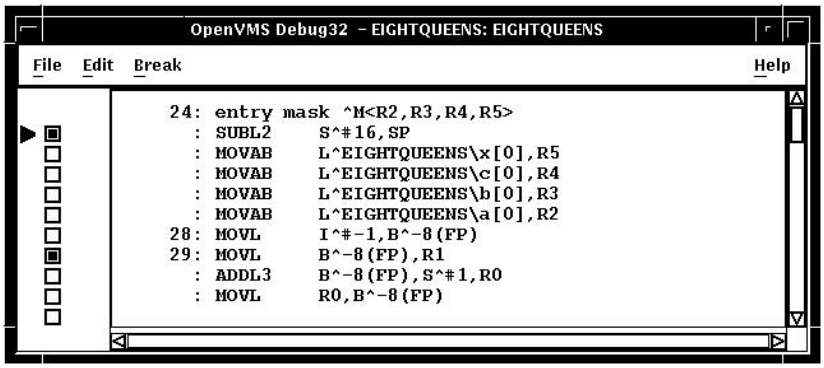
省略時の設定では，命令に対応するソース・コードの行番号が命令の左に表示されます。行番号を表示するか表示しないか指定するには，命令ビューの「File」メニューで「Display Line Numbers」を選択します。

省略時の設定では，命令の左にメモリ・アドレスが表示されます。アドレスを表示するか表示しないか指定するには，命令ビューの「File」メニューで「Show Instruction Addresses」を選択します。

命令ビューでの操作が終了したら，「Call Stack」メニューをクリックして，実行の停止箇所を再表示できます。

呼び出しスタック内のルーチンの命令ストリームを表示するには，メイン・ウィンドウの「Call Stack」メニューでそのルーチンの名前を選択します (第 10.6.2 項を参照)。

図 10–21 命令ビュー



10.9 タスキング (マルチスレッド) プログラムのデバッグ

タスキング・プログラム (マルチ・スレッド・プログラムとも呼ばれます) は 1 つのプロセス内に複数の実行スレッドを持っており，次のプログラムを含んでいます。

- DECthreads か POSIX 1003.1b サービスを使用する言語で書かれたプログラム。
- 言語固有のタスキング・サービス (その言語が直接用意しているサービス) を使用するプログラム。現在のところ，デバッガがサポートする組み込みタスキング・サービスを用意している言語は Ada だけである。

デバッガで使用する**タスク**や**スレッド**という用語はこのような制御の流れを示すものであり，言語や実現方法とは関係ありません。デバッガのタスキング・サポートは，このようなプログラムすべてに適用されます。

デバッガを使用すれば，タスクの情報を表示したり，タスクの実行，優先順位，状態の遷移などを制御するタスク特性を変更したりできます。

次の各節ではデバッガの DECwindows Motif for OpenVMS ユーザ・インタフェースのタスキング機能を要約します。デバッガのタスキング・サポートについての詳しい説明は，第 16 章を参照してください。

10.9.1 タスク (スレッド) 情報の表示

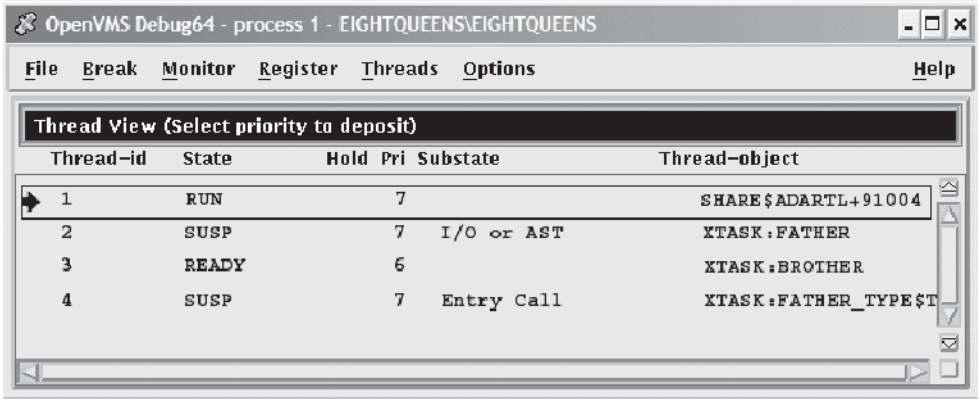
プログラムのタスク (スレッド) 情報を表示するには，メイン・ウィンドウまたはオプション・ビュー・ウィンドウの「Options」メニューで「Views...」を選択し，「Views」ダイアログ・ボックスが表示されたら，「Threads」をクリックします。

このスレッド・ビューには，ユーザ・プログラム内に現在存在している (終了していない) すべてのタスクの情報が表示されます。プログラムからデバッガに制御が戻ると，タスク情報が更新されます (図 10–22 を参照)。

次の各情報が表示されます。

- スレッド ID。左欄の矢印は，アクティブ・タスク，つまり「Go」ボタンや「Step」ボタンをクリックした時に実行されるスレッドを示す。
- スレッド優先順位。
- タスク (スレッド) が保留されているかどうか (第 10.9.2 項を参照)。
- タスク (スレッド) の現在の状態。「RUN」状態 (実行中) のタスクがアクティブ・タスクである。
- タスク (スレッド) の現在の副次状態。タスクが現在の状態になった原因を示す。

図 10–22 スレッド・ビュー



- タスク (スレッド) オブジェクトのデバッガ・パス名。デバッガがタスク・オブジェクトをシンボル化できない場合はタスク・オブジェクトのアドレス。

10.9.2 タスク (スレッド) 特性の変更

デバッグ中にタスク (スレッド) の特性やタスキング環境を変更するには，「Threads」メニューで次のいずれかの項目を選択します。

「Threads」メニューの項目	機能
Abort	選択されたタスク (スレッド) を次に終了可能なときに終了するように要求する。実際にどのように実行されるかは，現在のイベント機能により異なる (言語固有)。 Ada のタスクの場合，強制終了文の実行と同じである。
Activate	選択されたタスク (スレッド) をアクティブ・タスクにする。
Hold	選択されたタスク (スレッド) を保留にする。
Nohold	選択されたタスク (スレッド) の保留を解除する。
Make Visible	選択されたタスク (スレッド) を可視タスクにする。
All	サブメニューを使用してすべてのタスク (スレッド) を強制終了したり，すべてのタスク (スレッド) の保留を解除する。

10.10 デバッガの DECwindows Motif for OpenVMS インタフェースのカスタマイズ

デバッガは，省略時のデバッガ・リソース・ファイル (DECW$SYSTEM_DEFAULTS:VMSDEBUG.DAT) と一緒にシステムにインストールされます。このリソース・ファイルでは，次のようなカスタマイズ可能なパラメータで，省略時の起動設定を定義します。

- ウィンドウとビューの構成

- メイン・ウィンドウの行番号を表示するか表示しないかの指定
- ボタン名および対応するデバッガ・コマンド
- 条件付きブレークポイントとアクション・ブレークポイントのダイアログ・ボックスを表示するキー・シーケンス
- ソース・ビューと命令ビューにおける，テキストの言語依存の選択用キー・シーケンス
- ビューのテキストの文字フォント
- 各ウィンドウとビューに表示されるテキスト用の文字フォント
- ソース・ビュー，命令ビュー，エディタ・ビューにおける，テキストの前景色と背景色
- メイン・ウィンドウ・タイトル・バー上の，プログラム，モジュール，ルーチン名の表示
- デバッガを終了する前に確認が出るようにするかどうかの設定

説明用のコメントを入れた，システムの省略時デバッガ・リソース・ファイルは，Example 10–1 に掲載されています。これらの表示属性のうち，最初の 3 つの事項については，第 10.10.1 項，第 10.10.2 項，および第 10.10.3 項で説明する方法で DECwindows Motif for OpenVMS ユーザ・インタフェースにより会話形式で制御できます。どの場合も，「Options」メニューで「Save Options」を選択することにより，変更した表示構成を保存し，これ以降のデバッグ・セッションで適用することができます。

また，第 10.10.4 項の説明に従って，デバッガ・リソース・ファイルを編集し保存することにより，デバッガの表示構成についてのすべての属性を変更することができます。「Options」メニューから「Save Options」を選択するとき，またはローカル・デバッガ・リソース・ファイルの編集と保存を行うとき，デバッガは新しいバージョンのローカル・デバッガ・リソース・ファイル DECW$USER_DEFAULTS:VMSDEBUG.DAT を作成します。このファイルには，表示構成属性の定義が含まれています。次にデバッガを起動するとき，もっとも新しいローカル・リソース・ファイルで定義されている属性が使用され，それによって出力表示が構成されます。 DCL コマンドの DELETE， RENAME， COPY を適切に使用することにより，デバッガの以前の表示構成に戻すことができます。

システムの省略時表示構成に戻すときは， OpenVMS デバッガの「Options」メニューから「Restore Default Options」を選択します。

10.10.1 デバッガ・ビューの起動時構成の定義

デバッガのビューの起動時構成を定義するには，次の手順に従ってください。

1. デバッガの使用中に，ビューの構成を希望どおりに設定する。

2. 「Options」メニューで「Save Options」を選択することにより，デバッガ・リソース・ファイルの新バージョンが作成される。

次にデバッガを起動すると，新しい表示構成を作成するときに最新のリソース・ファイルが使用されるようになります。

リソース・ファイル内にあるこれらのビューの定義を編集することによって (第 10.10.4 項を参照)，起動時の表示構成を定義することもできます。

10.10.2 ソース・ビューと命令ビュー内の行番号の表示と非表示

デバッガの起動時，省略時の設定により，ソース・ビューと命令ビューにはソース行番号が表示されます。デバッガの起動時に行番号を表示しないようにするには，次の手順に従ってください。

1. デバッガの使用中に，メイン・ウィンドウ (または命令ウィンドウ) の「File」メニューで「Display Line Numbers」を選択する。そのメニュー項目の横のボタンが白抜きになり，行番号が表示されなくなる。
2. 「Options」メニューで「Save Options」を選択し，デバッガのローカル・リソース・ファイルの新バージョンを作成する。

次にデバッガを起動すると，最新のリソース・ファイルが使用されて，新しい表示構成が作成されます。

リソース・ファイル内の次のリソースを「True」か「False」に設定することによって (第 10.10.4 項を参照)，起動時に行番号を表示するかどうかの省略時設定を決めることもできます。

```
DebugSource.StartupShowSourceLineno: True
DebugInstruction.StartupShowInstLineno: True
```

10.10.3 プッシュ・ボタンの変更，追加，削除，並べ替え

プッシュ・ボタン・ビューのボタンはデバッガ・コマンドと対応しています。ユーザは次の操作ができます。

- ボタンのラベル，またはボタンに対応しているコマンドを変更する。
- 新しいボタンを追加する。
- ボタンを削除する。
- ボタンを並べ替える。

注意

「Stop」ボタンの変更，削除はできません。

これ以降のデバッガ・セッションで使用できるようにこれらの変更を保存するには，「Options」メニューで「Save Options」を選択します。

第 10.10.3.1 項，第 10.10.3.2 項，および第 10.10.3.3 項では， DECwindows Motif for OpenVMS ユーザ・インタフェースを使用して会話形式でプッシュ・ボタンをカスタマイズする方法について説明します。プッシュ・ボタンをカスタマイズするには，リソース・ファイルを編集する方法もあります。リソース・ファイルのボタン定義は，次のスクリプトで始まります。

```
DebugControl.Button
```

(Example 10–1 を参照。)

10.10.3.1 ボタンのラベルまたは対応するコマンドの変更

ボタンのラベルまたは対応しているコマンドを変更するには，次の手順に従ってください。

1. メイン・ウィンドウまたはオプション・ビュー・ウィンドウの「Options」メニューで「Customize Buttons...」を選択する。「Customize Buttons」ダイアログ・ボックスが表示される (図 10–23 を参照)。
2. 変更したいボタンをダイアログ・ボックスの中でクリックする。「Command」フィールドと「Label」フィールドにそのボタンのパラメータが表示される。図 10–23 では「STEP」ボタンを選択している。
3. ボタンのアイコンを変更するときは，ダイアログ・ボックスの「Icon」メニューをプルダウンして，定義済みアイコンを選択する。図 10–23 に示すように，「Label」フィールドの表示が薄くなり，定義済みアイコンの内部名が表示される。アイコン自体は，ダイアログ・ボックスのプッシュ・ボタン表示に表示される。

   ボタンのラベルを変更するときは，「Icon」メニューが「None」に設定されていることを確認してから，「Label」フィールドに新しいラベルを入力する。
4. ボタンに対応しているコマンドも変更するときは，「Command」フィールドに新しいコマンドを入力する。コマンドのオンライン・ヘルプについては，第 8.4.3 項を参照。

   ウィンドウで選択した名前や言語式にそのコマンドを作用させる場合は，コマンド・パラメータとして%Sを指定する。たとえば，次のコマンドでは現在選択されている言語式の現在値が表示される。

   ```
   EVALUATE %s
   ```

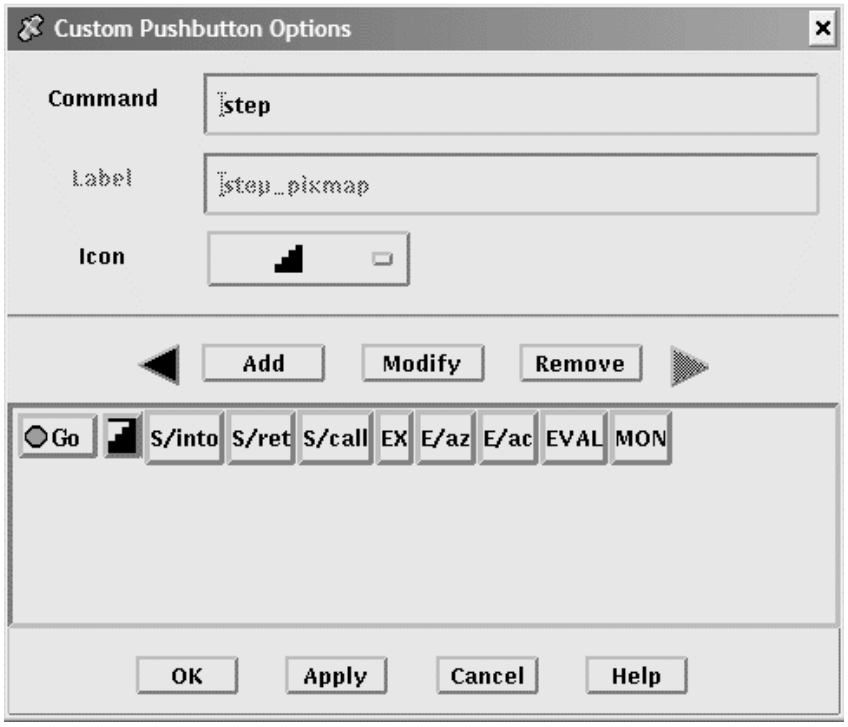
   デバッガの組み込みシンボル，またはパーセント記号(%)で始まるその他の名前にそのコマンドを作用させる場合は，次のように 2 つのパーセント記号を指定する。

   ```
   EXAMINE %%NEXTLOC
   ```

5. 「Modify」をクリックする。ダイアログ・ボックスのプッシュ・ボタン表示のボタン・ラベルまたは対応しているコマンドが変更される。
6. 「Apply」をクリックする。デバッガのプッシュ・ボタン・ビューのボタン・ラベルまたは対応しているコマンドが変更される。

これ以降のデバッガ・セッションで使用できるように変更事項を保存するときは，「Options」メニューから「Save Options」を選択します。

図 10–23 「Step」ボタン・ラベルのアイコンへの変更



10.10.3.2 新しいボタンおよび対応するコマンドの追加

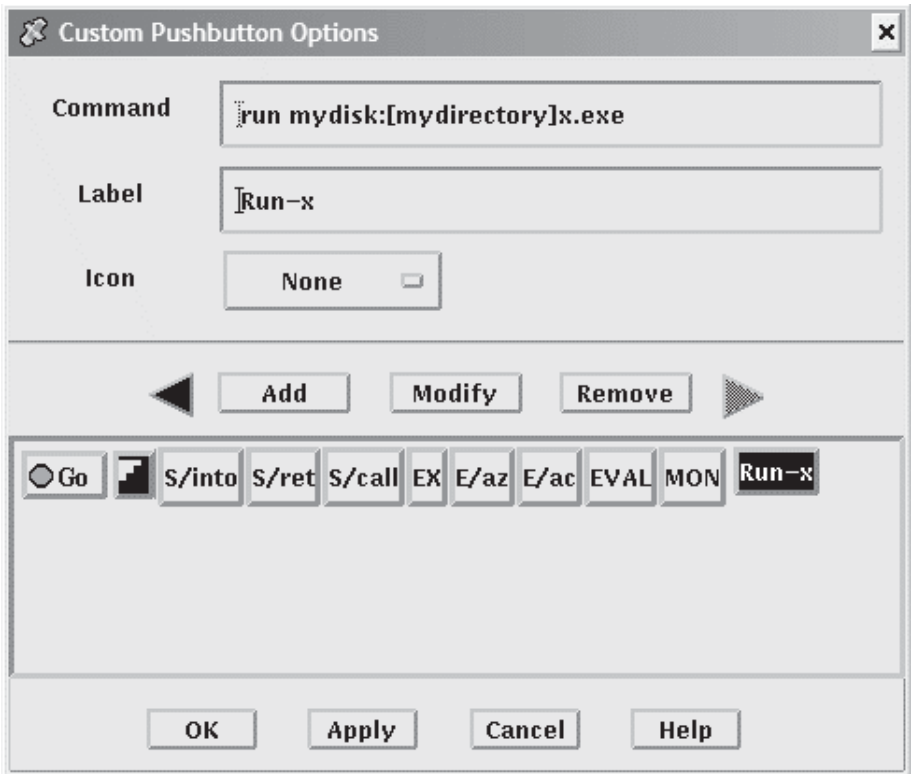
プッシュ・ボタン・ビューに新しいボタンを追加し，そのボタンにデバッガ・コマンドを割り当てるには，次の手順に従ってください。

1. 「Options」メニューで「Customize Buttons...」を選択する。「Customize Buttons」ダイアログ・ボックスが表示される (図 10–24 を参照)。
2. 「Command」フィールドに新しいボタンのデバッガ・コマンドを入力する (第 10.10.3.1 項を参照)。図 10–24 では，コマンド「RUN CP:X」を入力している。このコマンドは， X.EXE という名前のプログラムを起動する。

3. 「Label」フィールドにそのボタンのラベルを入力するか，「Icon」メニューの中から定義済みアイコンを選択する。図 10–24 では，「Run-X」ラベルを入力している。
4. 「Add」をクリックする。新しいボタンがプッシュ・ボタン表示のダイアログ・ボックスに追加される
5. 「Apply」をクリックする。デバッガのプッシュ・ボタン・ビューにボタンが追加される。

これ以降のデバッガ・セッションで使用できるように変更事項を保存するときは，「Options」メニューから「Save Options」を選択します。

図 10–24 「EXAMINE/ASCIZ」コマンドのボタンの追加



10.10.3.3 ボタンの削除

ボタンを削除するには，次の手順に従ってください。

1. メイン・ウィンドウまたはオプション・ビュー・ウィンドウの「Options」メニューで「Customize Buttons...」を選択する。「Customize Buttons」ダイアログ・ボックスが表示される。

2. 削除したいボタンをダイアログ・ボックスの中でクリックする。「Command」フィールドと「Label」フィールドにそのボタンのパラメータが表示される。
3. 「Remove」をクリックする。ダイアログ・ボックスのプッシュ・ボタン表示からボタンが削除される。
4. 「Apply」をクリックする。デバッガのプッシュ・ボタン・ビューからボタンが削除される。

これ以降のデバッガ・セッションで使用できるように変更事項を保存するときは，「Options」メニューから「Save Options」を選択します。

10.10.3.4 ボタンの並べ替え

ボタンを並べ替えるには，次の手順に従ってください。

1. メイン・ウィンドウまたはオプション・ビュー・ウィンドウの「Options」メニューで「Customize Buttons...」を選択する。「Customize Buttons」ダイアログ・ボックスが表示される。
2. 移動したいボタンをダイアログ・ボックスの中でクリックする。「Command」フィールドと「Label」フィールドにそのボタンのパラメータが表示される。
3. 右または左の矢印をクリックすると，選択したボタンの位置が 1 つずつ右または左に移動する。希望の位置に移るまで矢印のクリックを繰り返す。
4. 「Apply」をクリックする。設定した順序でデバッガのプッシュ・ボタン・ビューにボタンが表示される。

これ以降のデバッガ・セッションで使用できるように変更事項を保存するときは，「Options」メニューから「Save Options」を選択します。

10.10.4 デバッガ・リソース・ファイルの編集

デバッガは，省略時のデバッガ・リソース・ファイル (DECW$SYSTEM_DEFAULTS:VMSDEBUG.DAT) と一緒にシステムにインストールされます。このリソース・ファイルでは，デバッガの省略時表示構成を定義します。第 10.10 節の説明に従って表示属性を変更し，「Options」メニューの「Save Options」コマンドで変更を保存するとき，デバッガは，ローカル・デバッガ・リソース・ファイル DECW$USER_DEFAULTS:VMSDEBUG.DAT を作成します。このファイルを編集すると，デバッガの表示構成をさらに変更することができます。

ローカル・デバッガ・リソース・ファイルがない場合，「Options」メニューの「Restore Default Options」項目を選択することにより作成できるようになっています。最新バージョンのローカル・デバッガ・リソース・ファイルが 1 つある場合，デバッガを起動するときはいつでも，デバッガ表示構成がこのファイルで定義されているとおりに作成されます。それ以外の場合は，システム・デバッガ・リソース・ファイル DECW$SYSTEM_DEFAULTS:VMSDEBUG.DAT の定義が使用されます。

システム・リソース・ファイルは編集することができません。デバッガ表示構成は，第 10.10.1 項，第 10.10.2 項，第 10.10.3 項のいずれかの手順で変更することができます。またローカル・デバッガ・リソース・ファイルを編集してから保存しても，表示構成を変更できるようになっています。

Example 10–1 は，システムの省略時デバッガ・リソース・ファイルのコピーです。ほとんどのエントリには，ファイル内で注釈が付いているため，その意味が理解できるようになっています。第 10.10.4.1 項，第 10.10.4.2 項，第 10.10.4.3 項，第 10.10.4.4 項には，特定のキー・シーケンスを変更するときの注意が記載されています。キー・シーケンスの指定についての詳しい説明は，『X Toolkit Intrinsics』マニュアルの変換テーブルの構文を参照してください。

注意

Example 10–1 に記述されている行のうち，DebugControl.ButtonList**で始まるものは，この例に完全に合致するものではありません。この行は，ファイルに含まれているボタン定義を示しています。ファイル内の行には，**StepReturnButton，StepCallButton，ExamineButton，ExamineASCIZButton，ExamineASCICButton，EvalButton，MonitorButton**の各ボタン名も含まれています。**

Example 10–1 システムの省略時デバッガ・リソース・ファイル (DECW$SYSTEM_DEFAULTS:VMSDEBUG.DAT)

```
!
! OpenVMS Debug32/64 Debugger Resource File
!
DebugVersion: 71
!
! GEOMETRY RESOURCES:
!
! Written when you execute "SAVE OPTIONS" from the Options Menu.
!
DebugSource.x:   11
DebugSource.y:   30
DebugSource.width:  620
DebugSource.height:  700
!
DebugControl.x:   650
DebugControl.y:   30
DebugControl.width:  600
DebugControl.height:  700
!
DebugEditor.x:   650
DebugEditor.y:   30
DebugEditor.width:  600
DebugEditor.height:  700
!
DebugInstruction.x:  11
DebugInstruction.y:  769
DebugInstruction.width:  620
DebugInstruction.height: 243
!
*DebugBrowser.x:  650
*DebugBrowser.y:  30
*DebugBrowser.width:  335
*DebugBrowser.height:  300
```

(次ページに続く)

Example 10–1 (続き) システムの省略時デバッガ・リソース・ファイル (DECW$SYSTEM_DEFAULTS:VMSDEBUG.DAT)

```
!
! LINE NUMBER DISPLAY RESOURCES:
!
!   Create the line or address number display in views at startup?
!
DebugSource.StartupShowSourceLineno: True
DebugInstruction.StartupShowInstLineno: True
DebugInstruction.StartupShowInstAddrno: False
!
! WINDOW PANE RESOURCES:
!
! Relative size of panes in main window.
! Main window height is derived from sum of panes.
!
DebugSource*SrcView.height:  460
DebugSource*PushbuttonPanel.height: 36
DebugSource*MessageOutputPanel.height: 145
!
DebugControl.BreakpointView.height: 175
DebugControl.MonitorView.height: 150
DebugControl.TaskView.height:  130
DebugControl.RegisterView.height: 250
!
! CUSTOM BUTTON RESOURCES:
!
! The following resources determine which buttons to put in the button panel.
! Buttons will show in the order they are listed here.
! For each button there MUST be a set of associated resources.
! EXAMPLE:
!   ButtonCommand    - Associates a command with the button.
!   ButtonLegend     - Button Label or pixmap name if pixmap flag is True.
!   ButtonPixmapFlag - If True uses ButtonLegend as predefined pixmap name.
!
DebugControl.ButtonList: \ GoButton, StepButton, StepInButton, ...

!
DebugControl.ButtonCommand.GoButton:  go
DebugControl.ButtonLegend.GoButton:  go_pixmap
DebugControl.ButtonPixmapFlag.GoButton:  True
!
DebugControl.ButtonCommand.StepButton:  step
DebugControl.ButtonLegend.StepButton:  STEP
DebugControl.ButtonPixmapFlag.StepButton: False
```

(次ページに続く)

Example 10–1 (続き) システムの省略時デバッガ・リソース・ファイル (DECW$SYSTEM_DEFAULTS:VMSDEBUG.DAT)

```
!
DebugControl.ButtonCommand.StepInButton: step/in
DebugControl.ButtonLegend.StepInButton:  S/in
DebugControl.ButtonPixmapFlag.StepInButton: False
!
DebugControl.ButtonCommand.StepReturnButton: step/return
DebugControl.ButtonLegend.StepReturnButton: S/ret
DebugControl.ButtonPixmapFlag.StepReturnButton: False
!
DebugControl.ButtonCommand.StepCallButton: step/call
DebugControl.ButtonLegend.StepCallButton: S/call
DebugControl.ButtonPixmapFlag.StepCallButton: False
!
DebugControl.ButtonCommand.ExamineButton: examine %s
DebugControl.ButtonLegend.ExamineButton: EX
DebugControl.ButtonPixmapFlag.ExamineButton: False
!
DebugControl.ButtonCommand.ExamineASCIZButton: examine/asciz %s
DebugControl.ButtonLegend.ExamineASCIZButton: E/az
DebugControl.ButtonPixmapFlag.ExamineASCIZButton: False
!
DebugControl.ButtonCommand.ExamineASCICButton: examine/ascic %s
DebugControl.ButtonLegend.ExamineASCICButton: E/ac
DebugControl.ButtonPixmapFlag.ExamineASCICButton: False
!
DebugControl.ButtonCommand.EvalButton:  evaluate %s
DebugControl.ButtonLegend.EvalButton:  EVAL
DebugControl.ButtonPixmapFlag.EvalButton: False
!
DebugControl.ButtonCommand.MonitorButton: monitor %s
DebugControl.ButtonLegend.MonitorButton: MON
DebugControl.ButtonPixmapFlag.MonitorButton: False
```

(次ページに続く)

Example 10–1 (続き) システムの省略時デバッガ・リソース・ファイル (DECW$SYSTEM_DEFAULTS:VMSDEBUG.DAT)

```
!
! THE FOLLOWING RESOURCES CAN ONLY BE CHANGED BY EDITING THIS FILE.
! ----------------------------------------------------------------
! Be sure to trim off any trailing white-spaces.
!
! FONT RESOURCES:
!
!   If a font is specified for a view, and the font is available on the
!   system, it will be used for that view.
!
!   For any views which do not explicitly specify a font, the font specified
!   by the resource "DebugDefault.Font" will be used if it is available on the
!   system.
!
!   If no font resources are specified at all, the debugger will use the
!   systems own default font specification.
!
!   The "DebugOptions.Font" applies to all optional views.  We suggest that
!   you select a font with a point size no larger than 14 in the option views
!   in order to preserve label alignment.
!
!   Using 132 column sources?  Try this narrow font:
!       -dec-terminal-medium-r-narrow--14-100-100-100-c-60-iso8859-1
!
!              FORMAT:   -*-FONTNAM-FACE-T-*--*-PTS-*-*-*-*-CHARSET
!
DebugDefault.Font:  -*-COURIER-BOLD-R-*--*-120-*-*-*-*-ISO8859-1
DebugSource.Font:   -*-COURIER-BOLD-R-*--*-120-*-*-*-*-ISO8859-1
DebugInstruction.Font:   -*-COURIER-BOLD-R-*--*-140-*-*-*-*-ISO8859-1
DebugMessage.Font:  -*-COURIER-BOLD-R-*--*-120-*-*-*-*-ISO8859-1
DebugOptions.Font:   -*-COURIER-BOLD-R-*--*-120-*-*-*-*-ISO8859-1
!
! STARTUP RESOURCES: 3=Iconified, 0=Visible
!
DebugSource.initialState:  0
DebugControl.initialState:  0
DebugEditor.initialState:  0
DebugInstruction.initialState:  0
```

(次ページに続く)

Example 10–1 (続き) システムの省略時デバッガ・リソース・ファイル (DECW$SYSTEM_DEFAULTS:VMSDEBUG.DAT)

```
!
! COLOR RESOURCES:
!
! Use any of the OSF Motif Named Colors.
!
! Foreground = Text Color, Background = Window Color
!
! Try: Gainsboro, MintCream, Linen, SeaShell, MistyRose, Honeydew
!      Cornsilk, Lavender
!
! To use your system default color scheme, comment out all lines
! pertaining to color.
!
! Common color scheme (unless overridden for a specific view)
!
*background:     Gainsboro
*borderColor:     Red
!
! Source View Colors
!
!DebugSource*background:   Gainsboro
DebugSource*topShadowColor:                    WindowTopshadow
DebugSource*bottomShadowColor:                 WindowBottomshadow
DebugSource*src_txt.foreground:   blue
DebugSource*src_txt.background:   white
DebugSource*src_lineno_txtw.foreground:  red
DebugSource*cnt_msg_txt.foreground:  black
DebugSource*cnt_msg_txt.background:  white
!
! Control View Colors
!
!DebugControl*background:   Gainsboro
DebugControl*topShadowColor:                   WindowTopshadow
DebugControl*bottomShadowColor:                WindowBottomshadow
!
! Instruction View Colors
!
!DebugInstruction*background:   Gainsboro
DebugInstruction*topShadowColor:               WindowTopshadow
DebugInstruction*bottomShadowColor:            WindowBottomshadow
DebugInstruction*inst_txt.foreground:  blue
DebugInstruction*inst_txt.background:  white
DebugInstruction*inst_addrno_txtw.foreground:  red
```

(次ページに続く)

Example 10–1 (続き) システムの省略時デバッガ・リソース・ファイル (DECW$SYSTEM_DEFAULTS:VMSDEBUG.DAT)

```
!
! Editor Colors
!
!DebugEditor*background:   Gainsboro
DebugEditor*topShadowColor:                WindowTopshadow
DebugEditor*bottomShadowColor:             WindowBottomshadow
DebugEditor*edit_textw.foreground:  black
DebugEditor*edit_textw.background:  white
!
! REGISTER VIEW RESOURCES:
!

! Which Registers to display by default in the Register View?
! CF = Call Frame, GP = General Purpose, FP = Floating Point (Integrity and Alpha Only)

!
*Show_CF_Registers.set:  True
*Show_GP_Registers.set:  False
*Show_FP_Registers.set:  False
!
! SHOW MESSAGE/COMMAND SEPARATOR LINES?
!
*Show_Message_Separators.set:  True
!
! TRACK LANGUAGE CHANGES? (parser follows module language)
!
*Track_Language_Changes.set:  False
!
! KEY SEQUENCE RESOURCES:
!
! Key sequence used to activate the dialog box for conditional and action
! breakpoints.
!
DebugSource.ModifyBreakpointToggleSequence: Ctrl <Btn1Down>, Ctrl <Btn1Up>
!
! GENERAL KEYPAD FUNCTIONS:
!
!<Key>0xFFB0=KP0, <Key>0xFF91,<Key>0xFFB0=GOLD-KP0,
!<Key>0xFF94,<Key>0xFFB0=BLUE-KP0, <Key>0xFFB1=KP1,
!<Key>0xFF91,<Key>0xFFB1=GOLD-KP1, <Key>0xFFAC=KP,
DebugSource.*XmText.translations:#override\n\
    <Key>0xFFB0: EnterCmdOnCmdLine("step/line") \n\
    <Key>0xFFB1: EnterCmdOnCmdLine("examine") \n\
    <Key>0xFFAC: EnterCmdOnCmdLine("go") \n\
    <Key>0xFF91,<Key>0xFFB0: EnterCmdOnCmdLine("step/into") \n\
    <Key>0xFF94,<Key>0xFFB0: EnterCmdOnCmdLine("step/over") \n\
    <Key>0xFF91,<Key>0xFFB1: EnterCmdOnCmdLine("examine^") \n\
    <Key>0xFFB5: EnterCmdOnCmdLine("show calls") \n\
    <Key>0xFF91,<Key>0xFFB5: EnterCmdOnCmdLine("show calls 3") \n\
    <Key>0xFF8D: activate()\n
```

(次ページに続く)

Example 10–1 (続き) システムの省略時デバッガ・リソース・ファイル (DECW$SYSTEM_DEFAULTS:VMSDEBUG.DAT)

```
!
! IDENTIFIER WORD SELECTION: (language-based delimiters)
! NOTE: DO NOT use any double click combinitation for the following resource
!       otherwise normal text selection in the source window will not work.
!
DebugSource.IdentifierSelectionSequence: Ctrl<Btn1Down>
!
! EXIT CONFIRMATION:
!
DebugDisplayExitConfirmDB:  True
!
! COMMAND ECHO:
!
DebugEchoCommands:   True
!
! TITLE FORMAT: Main window and optional view window.
!
! The following title format directives are supported:
!
!   %t - The title of the debugger application.
!   %p - The name of the user program being debugged.
!   %f - The name of the current file displayed in the source window.
!
DebugControl.TitleFormat: %t - %p: %f
!
! DRAG AND DROP MESSAGE SUPRESSION: (Dont mess with these)
!
*.dragInitiatorProtocolStyle: DRAG_NONE
*.dragReceiverProtocolStyle:  DRAG_NONE
```

10.10.4.1 「Breakpoint」ダイアログ・ボックスを表示するキー・シーケンスの定義

省略時の設定では，条件付きブレークポイントとアクション・ブレークポイントのダイアログ・ボックスを表示するキー・シーケンスは， Ctrl/MB1 です (第 10.4.6 項と第 10.4.7 項を参照)。別のキー・シーケンスを定義するには，リソース・ファイル内の次のリソースの現在の定義を編集します。

```
DebugSource.ModifyBreakpointToggleSequence:   Ctrl<btn1down>(2)
```

10.10.4.2 言語依存のテキスト選択のキー・シーケンスの定義

省略時の設定では，メイン・ウィンドウと命令ウィンドウ内の言語依存のテキスト選択用のキー・シーケンスは， Ctrl/MB1 です (第 10.5.1 項を参照)。別のキー・シーケンスを定義するには，リソース・ファイル内の次のリソースの現在の定義を編集します。次に例を示します。

```
DebugSource.IdentifierSelectionSequence  Ctrl<btndown>
```

標準 DECwindows Motif for OpenVMS の単語選択との矛盾を避けるため，Ctrl<btn1down>(2) などのダブル・クリックとの組み合わせは使用しないでください。

10.10.4.3 表示テキスト用のフォントの定義

デバッガの各ウィンドウとビューに表示されるテキストに別のフォントを定義するには，リソース・ファイル内の次のリソースの現在の定義を編集します。次に例を示します。

```
DebugDefault.Font:  -*-COURIER-BOLD-R-*--*-120-*-*-*-*-ISO8859-1
```

10.10.4.4 キーパッド上のキーのバインディングの定義

既にコマンドを割り当てられているキーに別のコマンドをバインドするには，リソース・ファイル内の次のリソースの現在の定義を編集します。次に例を示します。

```
<key>0xFFB0: EnterCmdOnCmdLine("step/line 3") \n\
```

現在コマンドを割り当てられていないキーにコマンドをバインドする場合は，『X and Motif Quick Reference Guide』の Keysym Encoding の章のキー指定を参照してください。

10.11 独立プロセスのデバッグ

プリント・シンビオントのように，コマンド行インタプリタ (CLI) なしで動作する，独立プロセスをデバッグするとき，デバッガには DECwindows Motif for OpenVMS ユーザ・インタフェースを使用することはできません。

CLI なしで動作する，独立プロセスをデバッグするときは，デバッガに文字セル (画面モード) インタフェースを使用します (第 1.11 節を参照)。

第4部 PCクライアント・インタフェース

第 4 部では，デバッガの PC クライアント・インタフェースについて説明します。

デバッガのコマンド・インタフェースについては，第 2 部を参照してください。

デバッガの DECwindows Motif インタフェースについては，第 3 部を参照してください。

11

デバッガの PC クライアント/サーバ・インタフェースの概要

本章では，デバッガの PC クライアント・インタフェースについて説明します。

OpenVMS バージョン 7.3 デバッガは，以前のバージョンの PC クライアントはサポートしないことに注意してください。第 11.2 節で説明しているように， OpenVMS バージョン 7.3 のディストリビューション・メディアのキットに含まれているバージョン 1.1 の PC クライアントをインストールする必要があります。

バージョン 1.1 の PC クライアントには，バージョン 7.3 とそれ以前のデバッグ・サーバとの互換性があります。

11.1 概要

OpenVMS デバッガ・バージョン 7.2 およびそれ以降には，デバッガをデバッグ・クライアントとデバッグ・サーバの 2 つの構成要素に分割する，クライアント/サーバ・インタフェースというオプション機能があります。デバッグ・サーバは， OpenVMS システム上で実行されます。デバッグ・クライアントには， OpenVMS システム上で実行されるものと， Microsoft Windows，または Microsoft Windows NT 上で実行されるものがあります。どちらの Windows 環境でも，インタフェースは共通です。

PC クライアント・インタフェースには，実質的に DECwindows Motif クライアント・インタフェースと同じ機能が備わっています (第 9.9 節を参照)。

クライアントとサーバは，次のいずれかのトランスポート経由で， DCE RPC を使用して通信を行います。

- TCP/IP
- UDC
- DECnet

注意

TCP/IP Services for OpenVMS (UCX) バージョン 4.1 を実行している場合は，ECO2 がインストールされていることが必要です。UCX の最新バージョンを実行することもできます。

PC クライアント/サーバ・インタフェースを実行するために必要なソフトウェアについては，第 9.9.1 項を参照してください。

11.2 インストール

OpenVMS 上で実行される構成要素については，特別なインストール・プロシージャは必要ありません。本節では， PC 上のデバッグ・クライアントのインストール手順について説明します。

システム管理者は OpenVMS デバッガ・クライアント・キットを OpenVMS 配布メディアから PC ユーザがアクセスできるような， PATHWORKS シェアや FTP サーバに移す必要があります。次の表に， PC の構成に応じて，どのクライアント・キットを使用するかを示します。

CPU	オペレーティング・システム	クライアント・キット
Intel	Microsoft Windows 95, 98	[DEBUG_CLIENTS011.KIT]DEBUGX86010.EXE
Intel	Microsoft Windows NT	[DEBUG_CLIENTS011.KIT]DEBUGX86010.EXE
Alpha	Microsoft Windows NT	[DEBUG_CLIENTS011.KIT]DEBUGALPHA010.EXE

クライアント・キットは自動解凍型の.EXE ファイルになっています。

適切な実行ファイルを PC に転送したら，そのファイルを実行することによって，デバッグ・クライアントを PC 上にインストールできます。 INSTALLSHIELD インストール・プロシージャによって，インストールの手順が示されます。

省略時の設定では，デバッグ・クライアントは，`\Program Files\OpenVMS Debugger`フォルダにインストールされます。「Browse」をクリックして，別の場所を選択することもできます。

次のオプションのいずれかを選択することが可能です。

インストール・オプション	説明
Typical	デバッグ・クライアントと『 OpenVMS デバッガ説明書』の， HTML 形式のマニュアル。
Compact	デバッグ・クライアントのみ。
Custom	デバッグ・クライアントと『 OpenVMS デバッガ説明書』の， HTML 形式のマニュアルの一方，または両方。

Typical インストール・オプションを選択すると， OpenVMS Debugger のプログラム・フォルダに，次のものに対するショートカットが作成されます。

- デバッグ・クライアント

- デバッグ・クライアントのヘルプ・ファイル
- 『 OpenVMS デバッガ説明書』の HTML 形式のマニュアル
- Readme ファイル
- アンインストール・プロシージャ

11.3 プライマリ・クライアントとセカンダリ・クライアント

プライマリ・クライアントは，サーバに最初に接続されたクライアントです。セカンダリ・クライアントは，同じサーバに後から接続されたクライアントです。プライマリ・クライアントによって，サーバにセカンダリ・クライアントを接続できるようにするかどうかを制御することができます。

デバッグ・セッションに接続できるセカンダリ・クライアント数の指定の詳細については，第 11.5 節を参照してください。

11.4 PC クライアント・ワークスペース

PC クライアント・ワークスペースは， Motif クライアントのワークスペースと同様のものです (第 8 章を参照)。クライアント・ワークスペースには，動的な情報を表示するビューと，デバッガ・コマンドに対するショートカットが入ったツールバーが含まれています。必要に応じて，ビューとツールバーを設定したり，ショートカットを作成したり，設定を保存したりすることができます。

これらのトピックについては， PC クライアントのヘルプ・ファイルで詳しく説明されています。 PC クライアントのヘルプには， PC クライアントのインストール時に作成した OpenVMS Debugger フォルダ (第 11.2 節を参照) から直接アクセスするか，クライアントの「Help」メニューからアクセスします。次のトピックを参照してください。

- Overview
- Getting Started
- Views
- Toolbars

11.5 サーバ接続の確立

OpenVMS システムに直接ログインしてからデバッグ・サーバを起動することもできますが， eXcursion のような製品や Telnet のようなターミナル・エミュレータを使用して，リモートでログインするほうが便利です。

注意

デバッグ・サーバを実行するには，ライト・データベースに DBG$ENABLE_SERVER **識別子が必要です。デバッガ・サーバを使用するときにチェックしてください。デバッガ・サーバを実行すると，ネットワーク上の誰もがデバッガ・サーバに接続することができるようになります。**

DBG$ENABLE_SERVER 識別子を許可する前に，システム管理者はライト・データベースに書き込み (write) アクセスができるアカウントからDEBUG/SERVERコマンドを入力することによって識別子を作成しなければなりません。この作業を行うのは一度のみです。この作業の後，システム管理者は DBG$ENABLE_SERVER 識別子をユーザに許可するための Authorize ユーティリティを実行できます。

デバッグ・サーバを起動するには，次のコマンドを入力します。

```
$ DEBUG/SERVER
```

サーバのネットワーク・バインド文字列が表示されます。サーバのポート番号は，角括弧 ([ ]) で囲まれて表示されます。例を示します。

```
$ DEBUG/SERVER

%DEBUG-I-SPEAK: TCP/IP: YES, DECnet: YES, UDP: YES
%DEBUG-I-WATCH: Network Binding: ncacn_ip_tcp:16.32.16.138[1034]
%DEBUG-I-WATCH: Network Binding: ncacn_dnet_nsp:19.10[RPC224002690001]
%DEBUG-I-WATCH: Network Binding: ncadg_ip_udp:16.32.16.138[1045]
%DEBUG-I-AWAIT: Ready for client connection...
```

クライアントから接続する場合は，サーバを指定するために，いずれかのネットワーク・バインド文字列を使用します (第 9.9.4 項を参照)。

注意

通常は，ノード名とポート番号だけを使用してサーバを指定することができます。nodnam[1034]**がその例です。**

PC から接続を確立するには，「File」プルダウン・メニューで，または「Main」ツールバーの「C/S」ボタンを選択して，「Connection」ダイアログを表示します。ダイアログには，クライアントが認識しているサーバと，クライアントで現在アクティブになっているセッションが表示されます。

新規の接続についてサーバを指定したり，使用する特定のセッションを選択したりできます。

ダイアログ下部のボタンでは，次のことを行えます。

- 選択した (または省略時の設定の) サーバに接続する。
- サーバとの接続を切断する。

- クライアント/サーバ接続をテストする。
- 選択したサーバを停止する。

また，「Advanced」ボタンは使用するネットワーク・プロトコルを選択したり (第 11.5.1 項を参照)，確立するクライアント/サーバ接続で使用できるセカンダリ・クライアントの数 (0 ～ 30) を選択したりできます (第 11.5.2 項を参照)。

11.5.1 トランスポートの選択

「Connection」ダイアログでは，クライアント/サーバ接続に使用するネットワーク・プロトコルを，次から選択できます。

- TCP/IP
- DECnet
- UDP

11.5.2 セカンダリ接続

「Connection」ダイアログでは，サーバに接続するセカンダリ・クライアント (最大 30) を有効にすることができます。ただし，設定はプライマリ・クライアント (最初にサーバに接続したクライアント) から，次の手順に従って行ってください。

1. 「Connection」ダイアログで，「Advanced」をクリックする。
2. 「Properties」ダイアログで，使用できるようにするセカンダリ・クライアントの数を選択する (0 ～ 30)。
3. 「Connection」ダイアログの「Connect」をクリックする。

デバッガによって「Connection」ダイアログが消去されて，接続が確立されて，接続に成功 (または失敗) したことが「Command」ビューに表示されます。これで，デバッグ・プロシージャを開始することができます。

11.6 サーバ接続の終了

サーバを実行しているノード上で Ctrl-Y を入力すると，サーバを停止することができます。この方法でサーバを停止したら， DCL STOP コマンドを入力してください。

サーバをクライアントから停止するには，次の手順に従ってください。

1. 「File」プルダウン・メニューを開く。
2. 「Exit」を選択する。
3. サーバのみを停止する場合は「Server」をクリックする。サーバとクライアントの両方を停止する場合は，「Both」をクリックする。

次の手順でも，サーバを停止することができます。

1. 「File」プルダウン・メニューを開く。
2. 「Connection」をクリックして，「Connection」ダイアログを表示する。
3. 「Active Sessions」リストから，サーバ接続を選択する。
4. 「Stop」をクリックする。

11.6.1 クライアントとサーバの終了

サーバとクライアントの両方を停止するには，次の手順に従ってください。

1. 「File」プルダウン・メニューを開く。
2. 「Exit」をクリックする。
3. 「Both」をクリックする。

11.6.2 クライアントのみの終了

クライアントのみを停止するには，次の手順に従ってください。

1. 「File」プルダウン・メニューを開く。
2. 「Exit」をクリックする。
3. 「Client」をクリックする。

11.6.3 サーバのみの停止

サーバのみを停止するには，次の手順に従ってください。

1. 「File」プルダウン・メニューを開く。
2. 「Exit」をクリックする。
3. 「Server」をクリックする。

11.7 ドキュメント

PC クライアントのヘルプ・ファイルに加えて，『 OpenVMS デバッガ説明書』が HTML 形式で，オンラインで提供されます。クライアントからマニュアルにアクセスするには，次の手順に従ってください。

1. 「Help」プルダウン・メニューを開く。
2. 「Contents」をクリックする。
3. 「Local Manual」をクリックする。

第5部
高度なトピック

第 5 部では， OpenVMS デバッガが持つ高度なデバッグの技術について説明します。

12
ヒープ・アナライザの使用

ヒープ・アナライザはデバッガの 1 機能で，メモリの使用状況をリアルタイムにグラフィック表示します。ヒープ・アナライザは， OpenVMS Integrity および Alpha システムで使用できます。この表示を調べることで，ユーザ・アプリケーション内のどの領域でメモリの使用状況と性能を改善できるかを確認することができます。たとえば，頻繁に行われすぎる割り当て，大きすぎるメモリ・ブロック，フラグメンテーションの形跡，メモリ・リークなどを見つけることができます。

問題の領域がどこか見つけた後，ビューを拡大してさらに詳しく表示したり，変更したりすることができます。割り当てのサイズ，内容，アドレスなどの追加情報も表示することができます。

問題点を個々の割り当てにまで絞りこんだ後は，トレースバック情報を表示できます。ヒープ・アナライザでは，割り当てのトレースバック・エントリをアプリケーション・プログラムのソース・コードと相互に対応させることができます。それからソース・コード・ディスプレイをスクロールして調べていけば，問題のあるコードを特定でき，どう修正したらいいかを決定することができます。

本章では，次のことについて説明します。

- ヒープ・アナライザ・セッションの開始 (第 12.1 節)
- 省略時設定のディスプレイの作業 (第 12.2 節)
- タイプ設定とタイプ・ディスプレイの変更 (第 12.3 節)
- ヒープ・アナライザの終了 (第 12.4 節)
- サンプル・セッション (第 12.5 節)

12.1 ヒープ・アナライザ・セッションの開始

以下の各項では，ヒープ・アナライザを起動してユーザ・アプリケーションを実行する方法について説明します。

12.1.1 ヒープ・アナライザの起動

デバッグ・セッション中は，次のいずれかの方法でヒープ・アナライザを起動することができます。

1. デバッガのメイン・ウィンドウの「File」メニューから「Run Image」または「Rerun Same」を選択する。ダイアログ・ボックスが表示されたら，実行したいプログラムを選択して「Heap Analyzer」トグル・ボタンをクリックする。
2. デバッガのコマンド入力プロンプトから， RUN/HEAP_ANALYZER または RERUN/HEAP_ANALYZER program-imageコマンドを入力する。
3. Alpha システムでは，デバッガの外の DECterm ウィンドウで DCL プロンプト(＄)から次の各コマンドを実行してユーザ・プログラムを実行する。

   ```
   $ DEFINE/USER/NAME=CONFINE LIBRTL SYS$LIBRARY:LIBRTL_INSTRUMENTED
   ```

   保護されたイメージに対してヒープ・アナライザを使用するには，次のコマンドを入力して，プログラムを実行します。

   ```
   $ DEFINE/EXEC/NAME=CONFINE LIBRTL SYS$LIBRARY:LIBRTL_INSTRUMENTED
   ```

   この処理が必要なのは，次のコマンドを使用してイメージをインストールした場合です。

   ```
   $ INSTALL ADD imagename/PROTECTED
   ```

   ヒープ・アナライザは，デバッグ・セッションの外部からも起動できます。その場合には，上記の DEFINE/USER (または DEFINE/SYSTEM) コマンドを入力したあと， DCL の RUN/NODEBUG コマンドを入力します。
4. Integrity システムでは，デバッグ・プロンプト (DBG>) で， START HEAP_ANALYZER コマンドを入力する。

注意

アップコールが有効なスレッド・アプリケーションに対してヒープ・アナライザのスタートアップおよび Integrity サーバにおけるヒープ・アナライザの START コマンド (START HEAP_ANALYZER) を実行すると，デバッガはハングし続けます。

このような状況では，スレッド・アプリケーションあるいは AST 関連のアプリケーションに対しては，デバッグ・イベントを設定する前か，デバッグ・イベントを無効あるいは取り消した後にヒープ・アナライザを起動してください (ヒープ・アナライザのスタートアップ後にイベントを有効にあるいはリセットすることが可能で，START でデバッガの制御がユーザに戻ります)。

ヒープ・アナライザが正常に起動したら，スタートアップ画面が表示されます。

注意

OpenVMS Alpha **システムでは，**/NODEBUG **修飾子を使用してリンクしたプログラムで，ヒープ・アナライザは正しく動作しません。**

OpenVMS Integrity システムでは，/NODEBUG 修飾子を使用してリンクされたプログラムでヒープ・アナライザは動作しますが，表示されるトレースバック情報はわずかです。

12.1.2 ヒープ・アナライザのウィンドウ

ヒープ・アナライザには，メイン・ウィンドウ，6 つの補助ウィンドウ，およびコントロール・パネルがあります (図 12–1 を参照)。

最も重要なウィンドウである「Memory Map」には，ユーザ・アプリケーションによるメモリの使用状況が動的に表示されます。起動時の「Memory Map」には，アプリケーションを構成するイメージが表示されます。アプリケーションを実行すると，個々のメモリ・ブロック，イメージ，プログラム領域，メモリ・ゾーン，および動的文字列の相対記憶位置とサイズが，メモリ空間での割り当てと割り当て解除に応じて表示されます。

「Message」ウィンドウには，ヒープ・アナライザ・セッションについての情報が表示されます。起動時の「Message」ウィンドウには，'Heap Analyzer initialization complete. Press Start button to begin program' というメッセージが表示されます。ユーザ・アプリケーション実行中は，通報メッセージやエラー・メッセージが表示されます。

「Push Button」コントロール・パネルには，「Memory Map」ディスプレイの速度を調節するボタンがあります。デバッガを起動した後，ユーザ・アプリケーションの実行を開始するには，「Start」ボタンをクリックします。ユーザ・アプリケーション実行中にコントロール・パネルの他のボタンをクリックすると，連続表示の一時停止や低速化などの操作ができます。

「Information」ウィンドウには，「Memory Map」のセグメントについての情報が表示されます。アプリケーション実行中に，いつでも実行を一時停止して特定の情報を表示することができます。

「Source」ウィンドウには，「Memory Map」のセグメントに対応する，アプリケーションのソース・コードが表示されます。

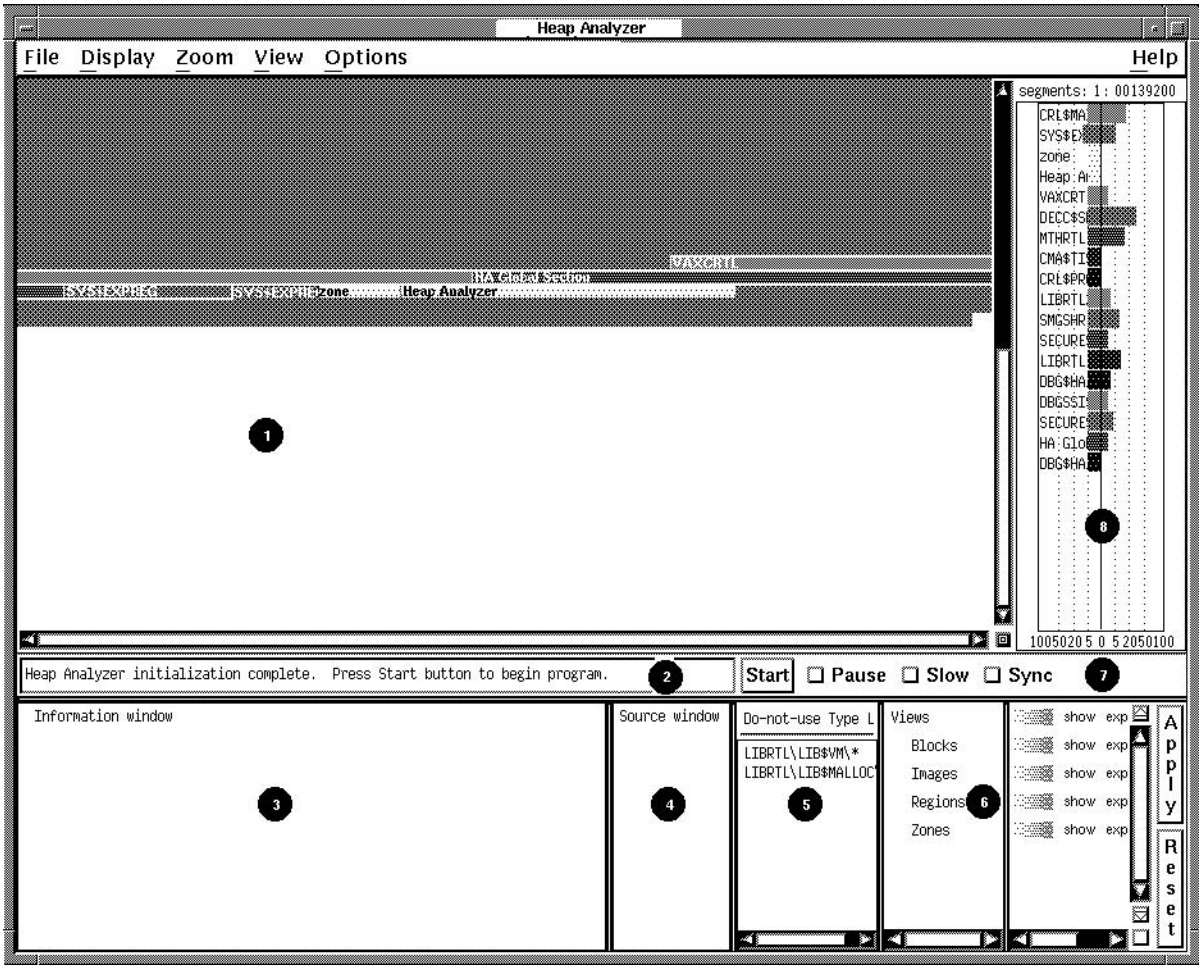
「Do-not-use Type」リストでは，セグメント・タイプつまりグループ名を再設定して，「Memory Map」ディスプレイを調整することができます。

「Views-and-Types」ディスプレイでは，選択した特定のセグメント・タイプを表示して，「Memory Map」ディスプレイを調整することができます。

「Type」ヒストグラムには，セグメント・タイプの要約と統計情報が表示されます。

ヒープ・アナライザを使用するとき，作業中のウィンドウのサイズを拡大または縮小しなければならない場合があります。この拡大や縮小には，ウィンドウ間の枠を引っぱるか，または画面全体のサイズを変更します。

図 12–1 ヒープ・アナライザのウィンドウ



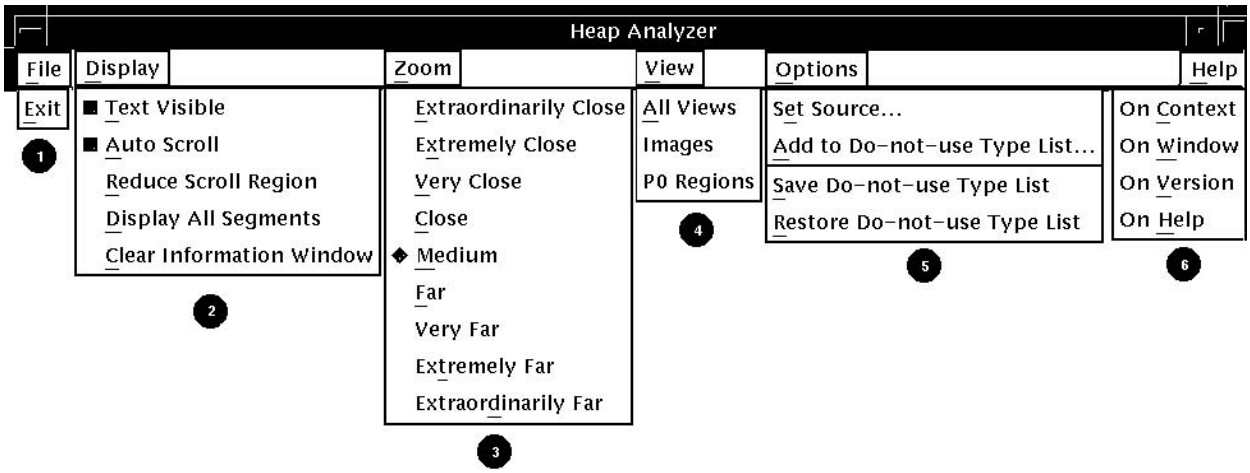
1.「Memory Map」	メモリつまり使用中の P0 空間部分をグラフィック表示する。それぞれの割り当ては色付きの帯のセグメントとして表示される。
2.「Message」ウィンドウ	ヒープ・アナライザの通報メッセージとエラー・メッセージやセグメントの説明を 1 行で表示する。
3.「Information」ウィンドウ	Memory Map」に表示されるセグメントとセグメント・タイプについての情報を表示する。
4.「Source」ウィンドウ	アプリケーションのソース・コードを表示する。
5.「Do-not-use Type」リスト	セグメント・タイプ (セグメントを特徴づける名前) として**使用しない**ルーチンの一覧を表示する。
6.「Views-and-Types」ディスプレイ	ヒープ・アナライザが認識しているセグメント・タイプの一覧を表示する。セグメント・ディスプレイの変更も行う。
7.「Push Button」コントロール・パネル	「Start」(「Step」), 「Pause」, 「Slow」, 「Sync」の各ボタンがある。「Memory Map」ディスプレイの速度を調節するためにボタンを使用する。

8.「Type」ヒストグラム　　セグメントのサイズと使用状況についての統計情報を表示する。

12.1.3 ヒープ・アナライザのプルダウン・メニュー

ヒープ・アナライザの「Memory Map」の上には 5 つのプルダウン・メニューがあります (図 12–2 を参照)。メニュー項目をすべて示すために，この図は少し修正してあります。

図 12–2 ヒープ・アナライザのプルダウン・メニュー

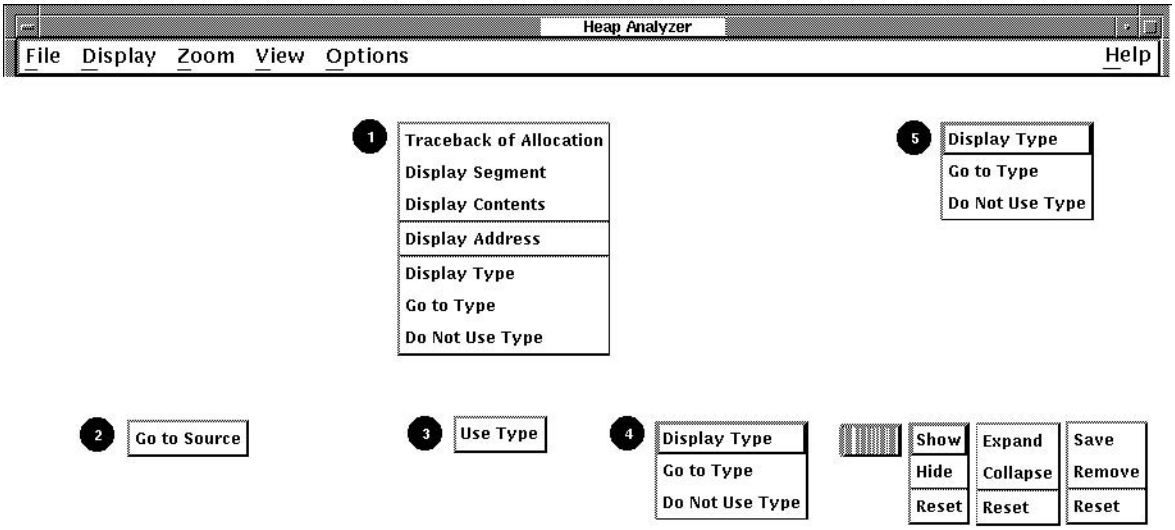


1.「File」メニュー　　ヒープ・アナライザを終了する。
2.「Display」メニュー　　「Memory Map」ディスプレイを調整したり，「Information」ウィンドウを消去する。
3.「Zoom」メニュー　　「Memory Map」のビューを拡大または縮小する。
4.「View」メニュー　　ディスプレイの粒度を選ぶ。
5.「Options」メニュー　　検索ディレクトリ・リストを指定したり，「Do-not-use Type」リストを修正する。
6.「Help」メニュー　　コンテキスト依存またはタスク指向のオンライン・ヘルプを表示する。

12.1.4 ヒープ・アナライザのコンテキスト依存のメニュー

ヒープ・アナライザのほとんどの操作は，コンテキスト依存のポップアップ・メニューから実行できます。ヒープ・アナライザのほとんどのウィンドウには，使用可能なタスクを並べたポップアップ・メニューがあります (図 12–3 を参照)。各ウィンドウのポップアップ・メニューにアクセスするには，そのウィンドウの中にマウス・ポインタを置いて MB3 をクリックします。

図 12–3 ヒープ・アナライザのコンテキスト依存のポップアップ・メニュー



1.「Memory Map」ポップアップ
「Memory Map」に表示されているセグメントについての追加情報を表示する。「Views-and-Types」ディスプレイにあるセグメント・タイプにジャンプする。また，「Do-not-use Type」リストにセグメント・タイプを追加する。

2.「Information」ウィンドウ・ポップアップ
「Information」ウィンドウに表示されているトレースバックの行から，「Source」ウィンドウにある対応したソース・コードへジャンプする。

3.「Do-not-use Type」リスト・ポップアップ
「Do-not-use Type」リストからセグメント・タイプを削除する。

4.「Views-and-Types」ディスプレイ・ポップアップ
左側: 表示されているセグメント・タイプについての追加情報を表示する。「Views-and-Types」ディスプレイ内でセグメント・タイプを強調表示する。また，「Do-not-use Type」リストにセグメント・タイプを追加する。

右側:「Views-and-Types」ディスプレイの左側で強調表示されたセグメント・タイプのディスプレイ属性を調整する。

5.「Type」ヒストグラム・ポップアップ
表示されているセグメント・タイプについての追加情報を表示する。「Type」ヒストグラム内でセグメント・タイプを強調表示する。また，「Do-not-use Type」リストにセグメント・タイプを追加する。

12.1.5 ソース・ディレクトリの設定

アプリケーションのソース・コードを格納しているディレクトリ以外のディレクトリからヒープ・アナライザを起動する場合は，起動画面が表示されてから，ヒープ・アナライザにソース・ディレクトリを設定することができます。

ソース・ディレクトリを設定するには，次の手順に従います。

1. ヒープ・アナライザ画面の「Options」メニューで「Set Source...」を選択する。

   「Set Source」ダイアログ・ボックスが表示される。

2. デバッガの SET SOURCE コマンドでの入力と同じように，ソース・ディレクトリのディレクトリ指定を入力する。

   SET SOURCE コマンドについての詳しい説明は，『デバッガ・コマンド・ディクショナリ』を参照。

3. 「OK」をクリックする。

これでヒープ・アナライザがユーザ・アプリケーションにアクセスできます。

12.1.6 アプリケーションの起動

デバッグ・セッションの中からヒープ・アナライザを起動した場合は，次の手順に従ってユーザ・アプリケーションを起動します。

1. 「Push Button」コントロール・パネルの「Start」ボタンをクリックする。

   「Message」ウィンドウに "application starting"というメッセージが表示され，「Start」ボタンのラベルが「Step」に変わる。 OpenVMS デバッガのメイン・ウィンドウが前面に出る。

2. デバッガのコントロール・パネルの「Go」ボタンをクリックしてから，OpenVMS デバッガのウィンドウをアイコン化する。

   ユーザ・アプリケーションに対応したメモリ・イベントの表示が，「Memory Map」内で開始される。

デバッグ・セッションの外でヒープ・アナライザを起動した場合は，上の手順の 1 だけを実行してユーザ・アプリケーションを起動してください。

アプリケーションが実行されると，「Memory Map」(および，ヒープ・アナライザの他の部分) は連続的に更新されて，ユーザ・アプリケーションの状態を反映します。

中断しなければ (第 12.1.7 項を参照)，この更新は，何らかのオカレンスによってメモリ・イベントが停止されるまで続きます。たとえば，ユーザ・アプリケーションが入力を求めるプロンプトを出す場合や，デバッガが入力を求めるプロンプトを出す場合，アプリケーションの実行が完了した場合などです。

12.1.7 表示速度の調節

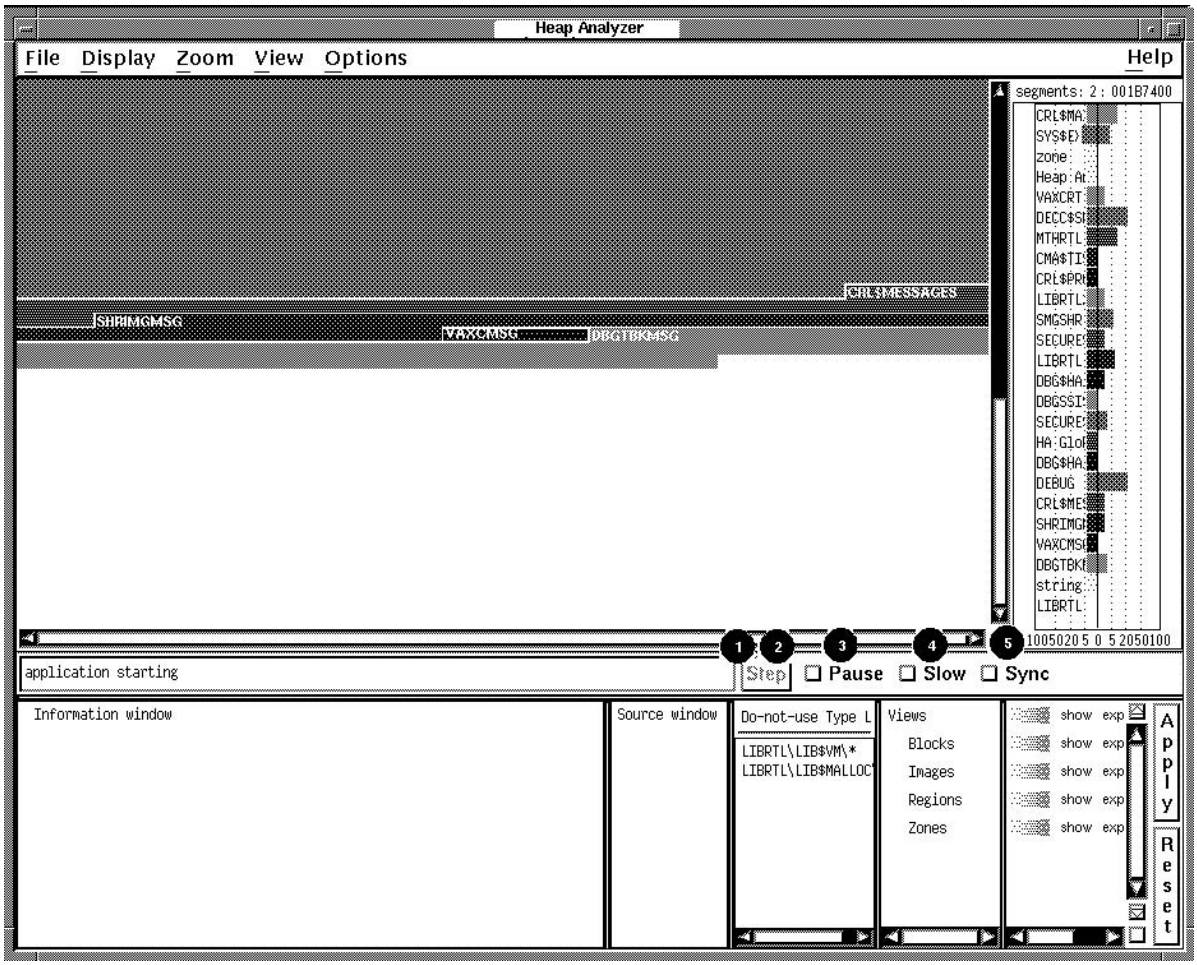
アプリケーションを実行しながら「Memory Map」でイベントを調べる場合，ヒープ・アナライザのプッシュ・ボタンを使用して，表示の一時停止，低速化などの速度調節が行えます。図 12–4 は，「Start」ボタンが押された直後のヒープ・アナライザ・ウィンドウの様子で，これらのプッシュ・ボタンがどう表示されるかを表しています。

「Slow」と「Pause」のプッシュ・ボタンは，それぞれ表示を低速化または一時停止します。

「Step」プッシュ・ボタンは，メモリ・イベントを 1 ステップずつ進めます。

「Sync」ボタンの右側の「Sync」ヒストグラム (図には示されていない) は，アプリケーションからどれだけ離れてヒープ・アナライザが実行されているかを表します。性能面での理由から，ヒープ・アナライザにメモリ・イベントが表示されるのは，アプリケーション内でそのイベントが発生してから数秒後になります。

図 12–4 ヒープ・アナライザのコントロール・パネル



1.「Start」ボタン　クリックすると，アプリケーションの実行と「Memory Map」ディスプレイが開始する。開始すると「Start」ボタンは「Step」ボタンに変わる。「Step」ボタンは初めは薄く表示されている (アクセスできない)。

2.「Step」ボタン　クリックすると，「Memory Map」ディスプレイ内のメモリ・イベントが 1 ステップずつ進むようになる。「Pause」ボタンをクリックするまでは薄く表示されている。

3.「Pause」ボタン　クリックすると，アプリケーションの実行と「Memory Map」の動的ディスプレイが一時停止する (または再開する)。

4.「Slow」ボタン　クリックすると，「Memory Map」の動的ディスプレイが低速になる。

5.「Sync」ボタン　クリックすると，ユーザ・アプリケーション・プログラムの実行と「Memory Map」内のメモリ・イベント・ディスプレイが同期する。

同時性が重要な場合，「Sync」プッシュ・ボタンを使用して，ヒープ・アナライザの表示とアプリケーションの実行とを同期させることができます。同期させるとアプリケーションの実行速度は低下します。

OpenVMS Alpha システムでは，デバッガやヒープ・アナライザのようにシステム・サービス・インタセプションを使用するものは，共有リンクによって起動されたシステム・サービス呼び出しイメージを受け取ることができません。そのためイメージを起動するプログラムは，イメージがリンクされているか/DEBUG を使って実行されている場合，共有リンクを避け，プライベート・イメージのコピーを起動するようにします。ただしこの場合，ヒープ・アナライザが制御するアプリケーションの性能に影響が現れ，共有リンクによって起動されたイメージほど高速に動作しなくなります。

12.2 省略時設定のディスプレイでの作業

以下の各項では，メモリに問題のあることが省略時設定の「Memory Map」ディスプレイを見て分かる場合に，ヒープ・アナライザをどう使用するかについて説明します。

見て分かる問題とは，割り当てが予想より大きすぎる，割り当てが頻繁に繰り返される，割り当てが行われるごとに割り当て量が増分される，より効率的な割り当てを行えることなどです。

そのような場合，ヒープ・アナライザ・セッションを次の手順で実行します。

1. 「Memory Map」ディスプレイを調べる。
2. 「Memory Map」のディスプレイ特性を設定する (省略可能)。
3. 個々のセグメントについての追加情報を求める (省略可能)。
4. 個々のセグメントについてのトレースバック情報を求める。
5. トレースバック・エントリをソース・コードのルーチンと相互に対応づける。

12.2.1 「Memory Map」ディスプレイ

「Memory Map」ディスプレイを調べるときは，ユーザ・アプリケーションのサイズに応じて，アプリケーション実行中に調査したい場合と実行完了後に調査したい場合とがあります。実行中は，プッシュ・ボタンを使用して，イベントを低速化または一時停止したり，1 ステップずつ進めたりできます。実行後は，「Memory Map」ディスプレイの縦のスクロール・バーを使用してディスプレイをスクロールできます。

サイズや記憶位置が期待と異なるセグメントを特定するには，「Memory Map」内のセグメントの位置が動的メモリ内のセグメントの記憶位置に対応していることを覚えておく必要があります。「Memory Map」ディスプレイの左上が動的メモリの最下位アドレスです。右へいくほど上位になり，表示は各行の右端から左端に折り返されます。

12.2.2 「Memory Map」ディスプレイのオプション

「Memory Map」を調べる場合，必要に応じてディスプレイ・オプションを選択すると，最も関心がある部分を分かりやすく表示することができます。

「Display」メニューでは，「Memory Map」内にセグメント・タイプ名を表示するかどうか，ディスプレイを自動的にスクロールして最新の動作を表示するかどうか，ディスプレイを圧縮するかどうかを指定できます。

「Zoom」メニューでは，「Memory Map」に表示されるセグメントの拡大率を指定できます。たとえば，「Far」メニュー項目を選択するとメモリを概観でき，「Extremely Close」を選択するとメモリの内容が詳細に表示されます。

図 12–5 に，「Display」プルダウン・メニューのディスプレイ・オプションを示します。この図には，「Memory Map」内で使用できるすべてのディスプレイ・オプションを示してあります。

図 12–5 ヒープ・アナライザの「Display」メニュー

1.「Display」メニュー

「Text Visible」(省略時の設定): 「Memory Map」の各セグメントにセグメント名のラベルを付ける (セグメントに名前のラベルを表示できるだけの大きさが必要である)。

「Auto Scroll」(省略時の設定): ディスプレイを拡大するとき，最上位のメモリ・アドレス (画面右下) に合わせて「Memory Map」を自動的にスクロールする。

「Reduce Scroll Region」:「Memory Map」ディスプレイを制限したり，部分的に表示する場合 (第 12.3.3.2 項)，元の表示位置をスクロールせずに，できるだけ多くのセグメントを表示できるようにディスプレイを圧縮する。

「Display All Segments」：全セグメントのセグメント定義を「Memory Map」に表示する。

「Clear Information Window」:「Information」ウィンドウのテキストとメッセージを消去する。

2.「Zoom」メニュー

「Memory Map」のビューを拡大または縮小するオプションを表示する。

12.2.3 詳細な情報についてのオプション

「Memory Map」ディスプレイを調べるとき，問題のありそうなセグメントについて，より多くの情報が必要になる場合があります。「Memory Map」ポップアップ・メニューでは，個々のセグメントのセグメント定義，内容定義，アドレス定義，およびタイプ定義を表示することができます。

セグメント定義の書式を次に示します。

```
cursor-address   n:init-address + length = end-address  name (view)
```

cursor-address	MB3 をクリックしたときのカーソル位置のアドレス
n	一連の全セグメント中のセグメントの序数
init-address	セグメントの初期アドレス
length	セグメント長 (バイト数)
end-address	セグメントの最終アドレス
name	セグメントのセグメント・タイプ名
view	セグメントのビュー (ブロック，イメージ，リージョン，またはゾーン。各ビューについての詳しい説明は，第 12.3.3.2 項を参照)

たとえば，次のセグメント定義は「Memory Map」内の 15 番目のセグメントを表します。このセグメントのタイプは LIBRTL です。

```
        0004ECA5     15: 00040000+0001CA00=0005CA00 LIBRTL (Image)
```

内容定義は，部分的なセグメント定義 (カーソル・アドレスのないセグメント定義) と，セグメント・アドレスの内容の ASCII 表現からなります。次に例を示します。

```
contents of: 38: 001C7000+000000C0=001C70C0
   LIBTRL\LIB$VM\LIB$GET_VM (Block)

          [ASCII representation]
```

アドレス定義は，指定されたアドレスへのユーザ・アクセスを文の形式で示します。次に例を示します。

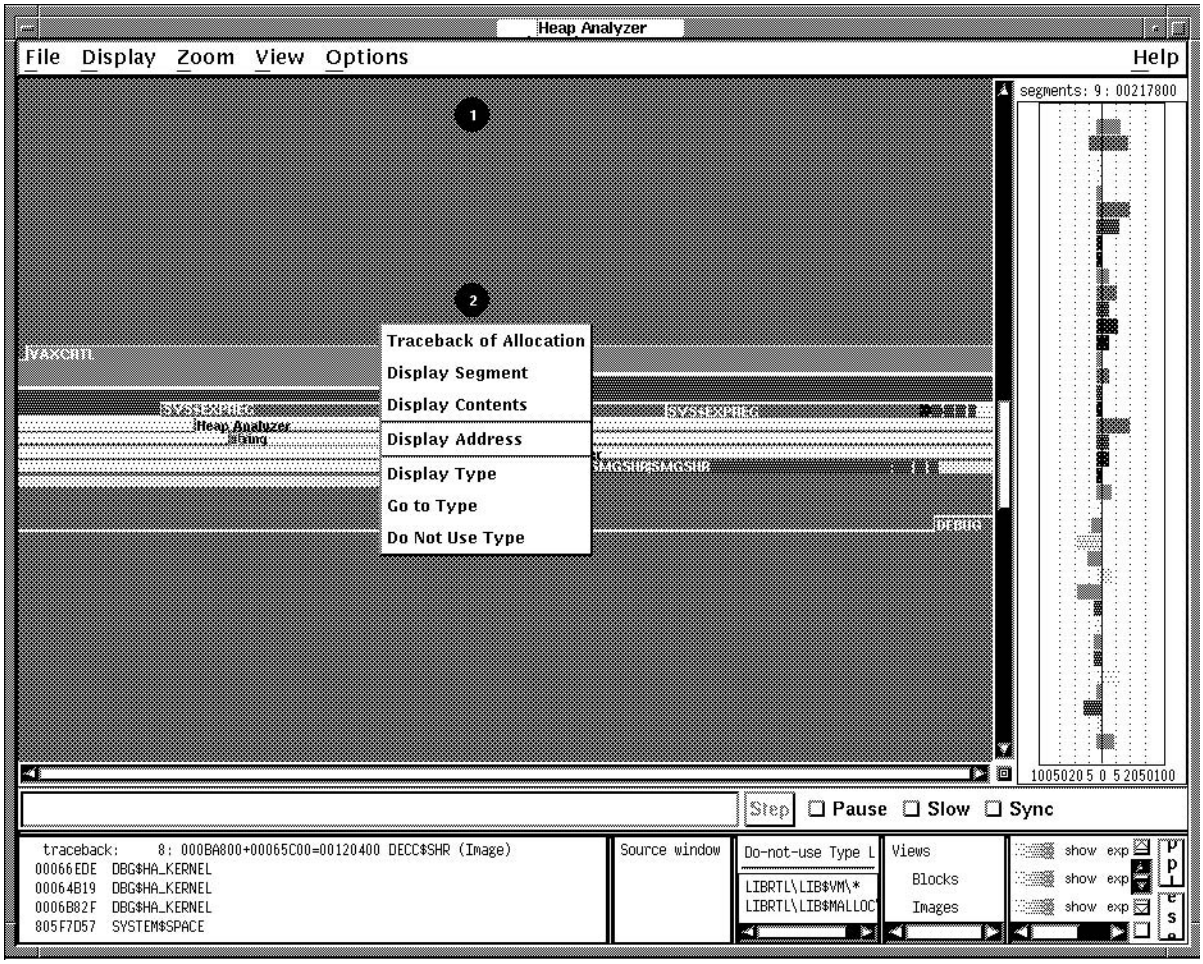
```
001C710B is read and write accessible by the user
```

タイプ定義は，セグメント・タイプが使用している総セグメント数と総バイト数を，要約した文の形式で示します。次に例を示します。

```
LIBRTL\LIB$VM\LIB$GET_VM (Block) has 39 segments
   using 00002160 bytes
```

図 12–6 に，コンテキスト依存の「Memory Map」ポップアップ・メニューを示します。この図には，「Memory Map」内で使用できるマウスとポップアップ・メニューのすべての項目を示してあります。

図 12–6 ヒープ・アナライザのコンテキスト依存の「Memory Map」ポップアップ・メニュー



1.「Memory Map」 MB1 のクリック:「Message」ウィンドウにセグメント定義を表示する。

2.「Memory Map」ポップアップ

「Traceback of Allocation」: セグメントに対応したトレースバック情報を「Information」ウィンドウに表示する (第 12.2.4 項を参照)。

「Display Segment」: セグメント定義を「Information」ウィンドウに表示する。

「Display Contents」: セグメント定義と各アドレスの内容を「Information」ウィンドウに表示する。

「Display Address」: カーソル位置のアドレスおよびユーザ・アクセスのタイプを「Information」ウィンドウに表示する。

「Display Type」: セグメント・タイプ定義を「Information」ウィンドウに表示する。

「Go to Type」:「Type」ヒストグラム内のセグメント・タイプから「Views-and-Types」ディスプレイにある同一のセグメント・タイプへジャンプする。

「Do Not Use Type」:「Do-not-use Type」リストにセグメント・タイプを追加する。

12.2.4 トレースバック情報の表示

問題のセグメントを個々に特定した後，「Memory Map」ポップアップ・メニューの「Traceback of Allocation」を選択します。トレースバック情報は，そのセグメントが作成された理由を知るのに役立ちます。トレースバックを調べることは，アプリケーション・コードを表示するための準備的な手順でもあります。

トレースバック情報は，部分的なセグメント定義 (カーソル・アドレスのないセグメント定義) と，セグメント作成時に呼び出しスタック上にあった要素のリストで構成されます。要素の命名規約は，**イメージ名\モジュール名\ルーチン名\行番号**です。次に例を示します。

```
traceback:     8:000BA800+00065C00=00120400 DECC$SHR (Image)
00066EDE   DBG$HA_KERNEL
00005864   CRL$MAIN_DB\CRL_LIBRARY\crl__initialize_libraries\%LINE 5592
```

12.2.5 トレースバック情報とソース・コードとの対応づけ

トレースバック・ディスプレイが現れたら，調べているセグメントに最も緊密に対応するトレースバック・エントリを探します。ほとんどの場合，セグメント・タイプ名とトレースバックのルーチン名とを比較することで見つけられます。

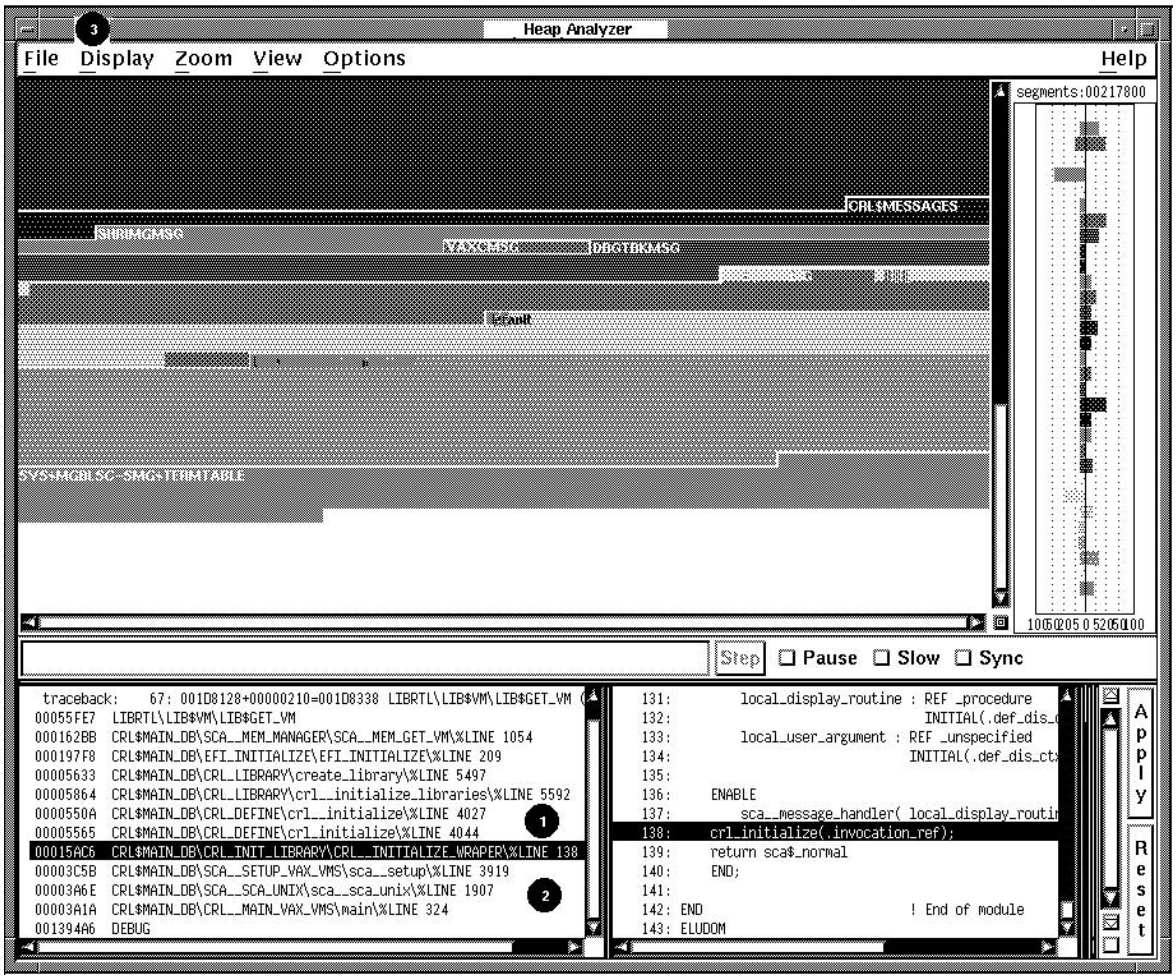
見つかったトレースバック・エントリ上で MB1 をダブル・クリックすると，そのエントリに対応したソース・コードが「Source」ウィンドウに (強調) 表示されます。次に，ソース・コード・ディスプレイをスクロールして，問題のあるコードを特定し，修正方法を決定することができます。

表示されたソース・コードに問題が見当たらない場合は，「Information」ウィンドウに戻り，前に選択したルーチンの直前または直後のルーチンを MB1 でダブル・クリックします。

トレースバック・エントリ上で MB1 をダブル・クリックしたときに 'Source Not Available' というメッセージが「Source」ウィンドウに表示された場合は，ヒープ・アナライザ・セッションの開始時にソース・ディレクトリの設定を忘れていた可能性があります。検索ディレクトリの設定については，第 12.1.5 項を参照してください。

図 12–7 に，「Information」ウィンドウ内で強調表示されているトレースバック・エントリを MB1 でダブル・クリックしたときに表示されるソース・コードを示します。この図には，「Source」ウィンドウと「Information」ウィンドウ内で使用できるマウスとメニューのすべての選択項目を示してあります。

図 12–7 ヒープ・アナライザの「Information」ウィンドウと「Source」ウィンドウ



1.「Information」ウィンドウ	MB1 のダブル・クリック:「Information」ウィンドウに表示されているトレースバックの行から「Source」ウィンドウ内の対応するソース・コードへジャンプできる。
2.「Information」ウィンドウ・ポップアップ	「Go to Source」:「Information」ウィンドウに表示されているトレースバックの行から「Source」ウィンドウ内の対応するソース・コードへジャンプできる。
3.「Display」メニュー	「Clear Information Window」:「Information」ウィンドウのテキストやメッセージを消去する。

12.3 タイプ設定とタイプ・ディスプレイの変更

以下の各項では，省略時設定の「Memory Map」に表示されるメモリ・イベントが分かりにくく，問題の有無を判断できない場合に実行する手順について説明します。

こういう状況になるのは，ヒープ・アナライザが選択するセグメント・タイプ名が多すぎてユーザ・アプリケーションには不都合なときや，「Memory Map」内の表示が多すぎて問題のセグメントを見分けにくいときです。

このようなときは，次の 1 つまたはいくつかの方法を選択できます。

- 「Type」ヒストグラムでタイプの要約を調べる (総セグメント数と総バイト数に占める各セグメント・タイプの使用率を示した要約を調べる)。
- 「Memory Map」内のタイプ設定を変更する (意味のあるタイプ名を選択することを指定する)。
- 「Memory Map」内のタイプ・ディスプレイを変更する (特定のタイプ・ディスプレイを抑制し，他のタイプを強調表示することを指定する)。

タイプ設定またはタイプ・ディスプレイを変更することで問題点が明らかになったら，省略時の「Memory Map」ディスプレイで作業する場合と同じ方法で問題を解決できます (詳しい説明は，第 12.2 節を参照してください)。

12.3.1 詳細な情報についてのオプション

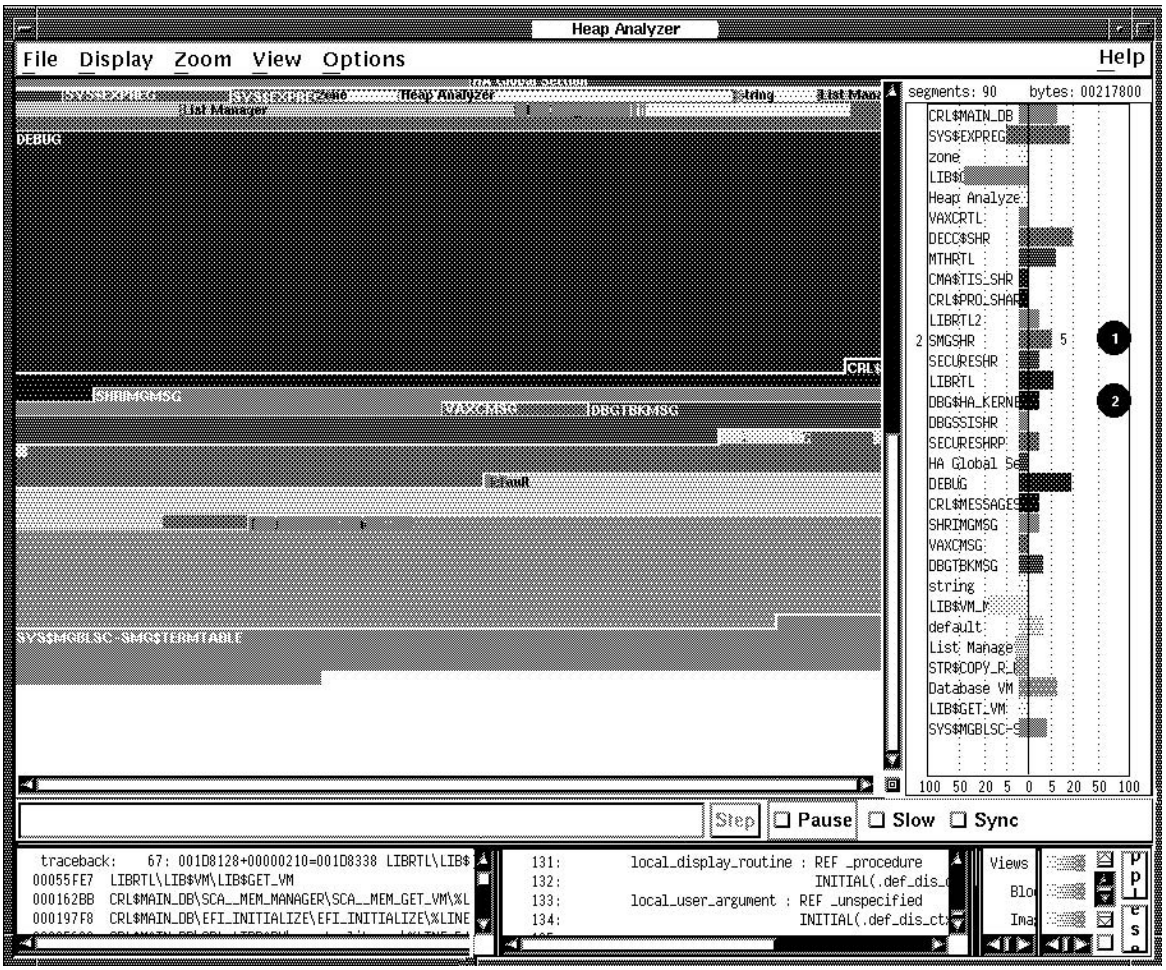
「Memory Map」を調べるとき，「Memory Map」の動きを要約した「Type」ヒストグラムを同時に参照したい場合があります。 2 つのヒストグラムを背中合わせにした「Type」ヒストグラムには，「Memory Map」の各セグメント・タイプが，総セグメント数と総バイト数の何パーセントを占めているかが表示されます。

グラフィック表現を数値に変えるには，それぞれのセグメント・タイプを MB1 でクリックします。

総セグメント数と総バイト数は両ヒストグラムの上部に表示されます。

図 12–8 は，「Type」ヒストグラムに表示されたセグメント・タイプを示しています (図のウィンドウは全タイプを表示するためにサイズを変更しています)。この図には，「Type」ヒストグラム内で使用できるマウスとメニューのすべての選択項目を示してあります。

図 12–8 ヒープ・アナライザの「Type」ヒストグラム



1.「Type」ヒストグラム

MB1 のクリック: 総セグメント数と総バイト数に占める各セグメントの使用率を表示する。

2.「Type」ヒストグラム・ポップアップ

「Display Type」: タイプ定義を「Information」ウィンドウに表示する。

「Go to Type」:「Type」ヒストグラム内のセグメント・タイプから「Views-and-Types」ディスプレイにある同一のセグメント・タイプへジャンプする。

「Do Not Use Type」:「Do-not-use Type」リストにセグメント・タイプを追加する。

12.3.2 タイプ設定の変更

「Memory Map」を調べると，ユーザには無意味なセグメント・タイプ名が見つかる場合があります。このような名前を「Do-not-use Type」リストに追加することで，セグメント名の変更と，必要な場合には「Memory Map」ディスプレイの再生成をヒープ・アナライザに指示します。

省略時の設定では，セグメント・タイプ名はセグメント作成時に割り当てられます。場合によっては要素の名前 (たとえば， LIBRTL) が割り当てられることもありますが，ほとんどの場合，ヒープ・アナライザは呼び出しスタックを下方向へ検索してルーチン名を探し，そのルーチン名をセグメント・タイプ名として使用します。

アナライザが選択するルーチン名は，呼び出しスタック内にあり，「Do-not-use Type」リストによって禁止されていない最初のルーチン名です。先頭のルーチンが禁止されている場合は，順に次のルーチンを調べていきます。

省略時のこの動作は，「Memory Map」に次の問題を引き起こすことがあります。

- 何種類かの同一のタイプ名が「Memory Map」ディスプレイに繰り返し表示される。

  呼び出しスタックの最初のルーチンが，下位レベルのメモリ管理ルーチンまたはユーティリティ・ルーチンである場合に起きる。ユーザ・アプリケーションの割り当てイベントの大部分はこれらのルーチンを使用するので，関連のない割り当てが同じタイプ名で 1 つにグループ化されて表示される。

  この問題を防ぐには，ユーザ・アプリケーションを実行する前に，アプリケーション固有のメモリ管理ルーチンまたはユーティリティ・ルーチンの名前を「Do-not-use Type」リストに追加しておく。

- 割り当てられたタイプ名による抽象化レベルが，ユーザに必要なレベルよりも高くなる。

  呼び出しスタックの最初のルーチンが，ユーザの調べているレベルほどのアプリケーション・バウンドでない場合に起きる可能性がある。アプリケーションの各関数を反映したタイプ名を見ることが必要な場合は，中間のメモリ管理ルーチンに由来するタイプ名が表示されるとかえって不便になる。

  呼び出しスタックの最初のルーチンがユーザ・アプリケーションの一部分を対象にしていて，ユーザがその部分に関心がない場合も，この問題が起きる可能性がある。サブシステムの関数 (たとえば， initialize_death_star) を反映したタイプ名を見ることが必要な場合，サブシステムの全関数 (たとえば， initialize_star) が 1 つのタイプ名で表示されては不便になる。

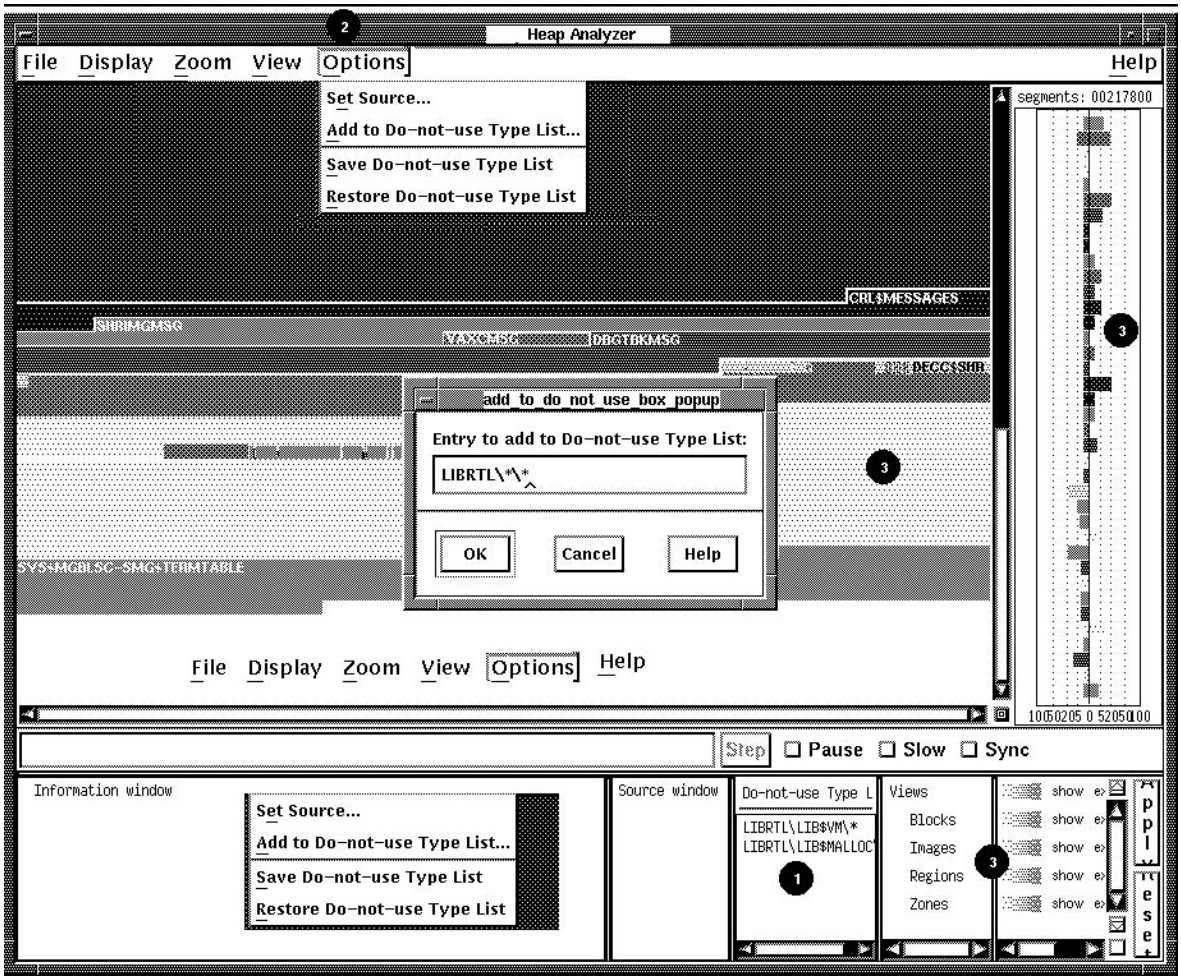
  この問題を修正するには，「Memory Map」に反映される抽象化レベルがユーザの希望と一致するまで，現在のタイプ名を「Do-not-use Type」リストに追加していく。

セグメント・タイプ名を「Do-not-use Type」リストに追加するには，プルダウンの「Options」メニューから「Add to Do-not-use Type List」を選択するか，または「Memory Map」，「Type」ヒストグラム，および「Views-and-Types」ディスプレイの各ポップアップ・メニューで「Do Not Use Type」を選択します。「Do-not-use Type」リストからセグメント・タイプを削除するには，「Do-not-use Type」リストのポップアップ・メニューで「Use Type」を選択します。

「Do-not-use Type」リストの内容を保存するには，「Options」メニューで「Save Do-not-use Type」リストを選択します。今後のヒープ・アナライザ・セッション用にリストの内容が保存されます。メニュー項目の「Restore Do-not-use Type」リストを使用すると，リストの最後の保存後の追加分が削除されます。

図 12–9 に，「Add to Do-not-use Type」リスト・ダイアログ・ボックスへの LIBRTL **の入力を示します。このダイアログ・ボックスは「Options」メニューから選択できます。この図には，「Do-not-use Type」リスト内で使用できるマウスとメニューのすべての選択項目を示してあります。

図 12–9 ヒープ・アナライザの「Do-not-use Type」リスト



1.「Do-not-use Type」リスト・ポップアップ	「Use Type」:「Do-not-use Type」リストからセグメント・タイプを削除する。
2.「Options」メニュー	「Add to Do-not-use Type List」:「Do-not-use Type」リストにセグメント・タイプを追加する。
	「Save Do-not-use Type List」:「Do-not-use Type」リストに表示されているセグメント・タイプを保存する。次のヒープ・アナライザ・セッションで使用できる。
	「Restore Do-not-use Type List」:「Do-not-use Type」リストを最後に保存してからの追加分を削除する。
3.「Memory Map」ポップアップ「Type」ヒストグラムポップアップ「Views-and-Types」ディスプレイ・ポップアップ	「Do Not Use Type」:「Do-not-use Type」リストにセグメント・タイプを追加する。

12.3.3 「Views-and-Types」ディスプレイの変更

「Memory Map」を調べるとき，重要な領域がはっきりするようにタイプ・ディスプレイを変更しなければならない場合があります。「Views-and-Types」ディスプレイでは，同じタイプの複数または個々のセグメントに対して変更を指定できます。

「Views-and-Types」ディスプレイは実際は 2 つのウィンドウからなり，枠で区切られています。左のウィンドウは拡大して，ユーザ・アプリケーション内の既知のタイプをすべて表示できます。右のウィンドウにはディスプレイ・オプション (色，表示状態，拡大状態，保存状態) が表示されます。

12.3.3.1 変更の有効範囲の選択

ヒープ・アナライザは OpenVMS の 4 つのメモリ・マネージャからセグメントについての情報を受け取ります。これらのメモリ・マネージャは，メモリ空間の割り当てと割り当て解除を実行します。各メモリ・マネージャが持つ動的メモリのビュー (全体像) は少しずつ異なっています。

メモリ・マネージャが認識するセグメント・タイプのセットは，マネージャごとに異なります。これは，各メモリ・マネージャからのビューが互いに重なっている部分では，ヒープ・アナライザ内で，単一のメモリ記憶位置が 1 つまたは複数のセグメント・タイプと対応する場合があることを意味します。

「Views-and-Types」ディスプレイの左のウィンドウには，この重なりを反映した階層が表示されます。

- 「Views」(4 つのビューをすべて統合する)
- 「Blocks」(LIB$VM メモリ・マネージャからのブロック・ビュー)
- 「Images」(SYS$IMAGE メモリ・マネージャからのイメージ・ビュー)
- 「Regions」(SYS$SERVICES メモリ・マネージャからのシステム・サービス・ビュー)

- 「Zones」(LIB$VM_ZONE メモリ・マネージャからのゾーン・ビュー)

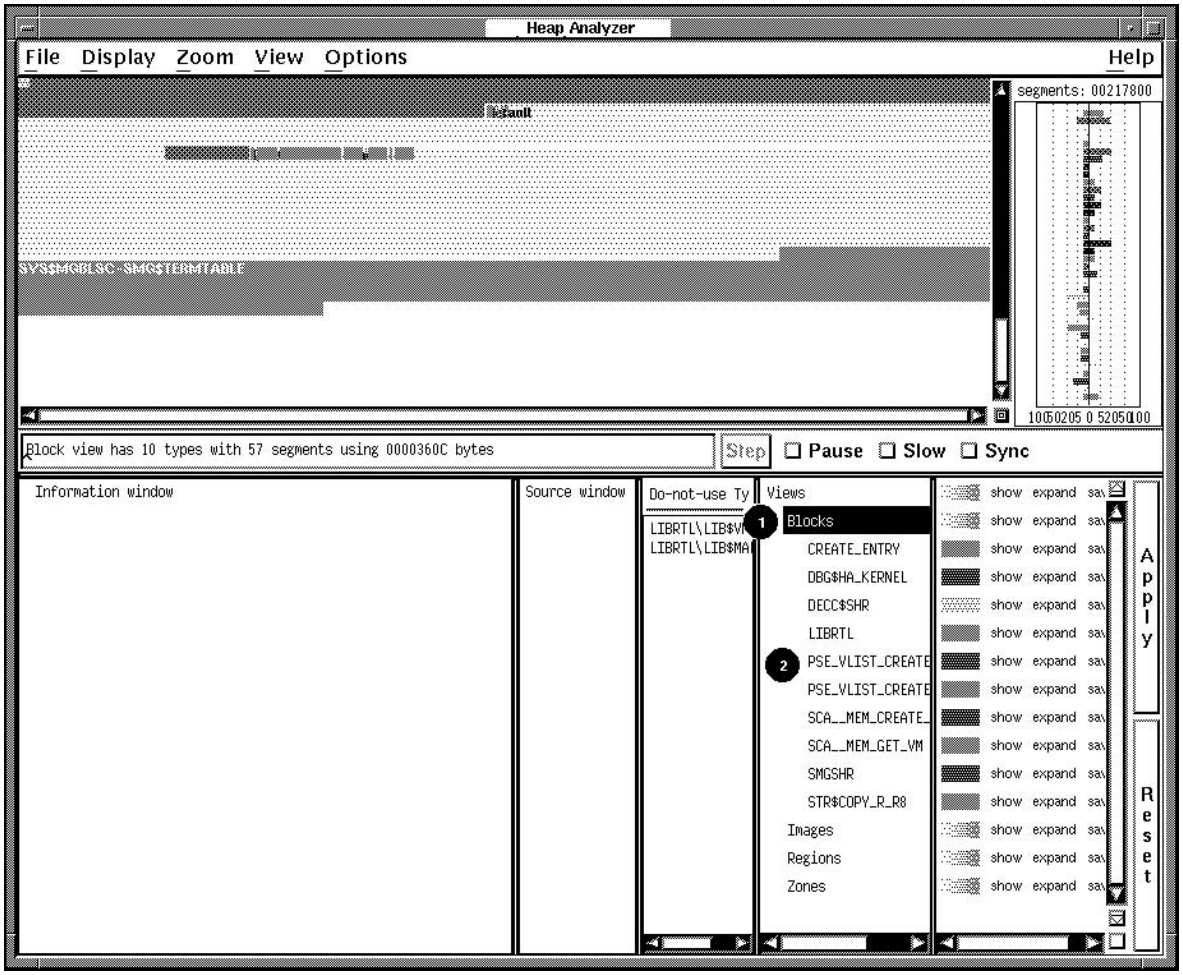
各メモリ・マネージャが認識している個々のセグメント・タイプを表示するには，「Blocks」，「Images」，「Regions」，「Zones」の各キーワードを MB1 でダブル・クリックして，省略時設定の表示を拡大します。それぞれの表示を元に戻すには，前に選択したキーワードを MB3 でクリックします。

この階層で次の有効範囲を選択できます。

- すべてのビューの全セグメント・タイプに作用させるには，「Views」キーワードを MB1 でクリックする。
- 1 つのビューの全セグメント・タイプに作用させるには，「Blocks」，「Images」，「Regions」，「Zones」の各キーワードを MB1 でダブル・クリックする。
- 個々のセグメント・タイプに作用させるには，選択するビューを MB1 でダブル・クリックしてから， 1 つまたは複数の単一のセグメント・タイプを MB1 でクリックする。

図 12–10 に，「Blocks」の階層の項目を示します。「Blocks」を MB1 でクリックして全ブロックを選択すると，「Blocks」が強調表示されます。この図には，「Views-and-Types」ディスプレイの階層側で使用できるマウスとメニューのすべての選択項目も示してあります。

図 12–10 ヒープ・アナライザの「Views-and-Types」の階層



1. MB1 のダブル・クリック	「Views-and-Types」の階層を拡大 (または縮小) する。
2.「Views-and-Types」階層のポップアップ	「Display Type」: タイプ定義を「Information」ウィンドウに表示する。 「Go to Type」:「Views-and-Types」ディスプレイ内で選択したタイプを強調表示する。 「Do Not Use Type」:「Do-not-use Type」リストにセグメント・タイプを追加する。

12.3.3.2 ディスプレイ・オプションの選択

「Views-and-Types」ディスプレイの右ウィンドウには，使用可能な次のディスプレイ・オプションがあります。

- 色

  全セグメント・タイプ，特定のビューだけの全セグメント・タイプ，または個々のセグメント・タイプの色を変更するには，「Views-and-Types」ディスプレイの色のボタンを MB3 でクリックする。縦の色の帯が表示されたら，選択する色を MB1 でクリックする。その後「Apply」ボタンをクリックして，変更を実際に適用する。

- 表示 (または表示しない) 状態

  全セグメント・タイプ，特定のビューだけの全セグメント・タイプ，または個々のセグメント・タイプのディスプレイを抑制 (または復元) するには，「Show」ボタンを選択して設定を「Hide」または「Show」に切り替えてから MB1 をクリックする (または，「Show」ポップアップ・メニューで適切なメニュー項目を選択する方法もある)。その後「Apply」ボタンをクリックして，変更を実際に適用する。

  調べないセグメントの「Memory Map」を消去する場合に，このオプションを使用する。このオプションを使用して，他のタイプの全セグメントを表示しないようにすれば，特定のタイプのセグメントをすべて見つけることもできる。

- 拡大 (または縮小) 状態

  全セグメント・タイプ，特定のビューだけの全セグメント・タイプ，または個々のセグメント・タイプのディスプレイを縮小 (または拡大) するには，「Expand」ボタンを選択して設定を「Collapse」または「Expand」に切り替えてから MB1 をクリックする (または，「Expand」ポップアップ・メニューで適切なメニュー項目を選択する方法もある)。その後「Apply」ボタンをクリックして，変更を実際に適用する。

  ネストしたセグメントは調べないので，そのようなセグメントの「Memory Map」を消去する場合に，このオプションを使用する。ユーザ・アプリケーションによっては，ヒープ・アナライザの性能も向上することがある。

- 保存 (または削除) 状態

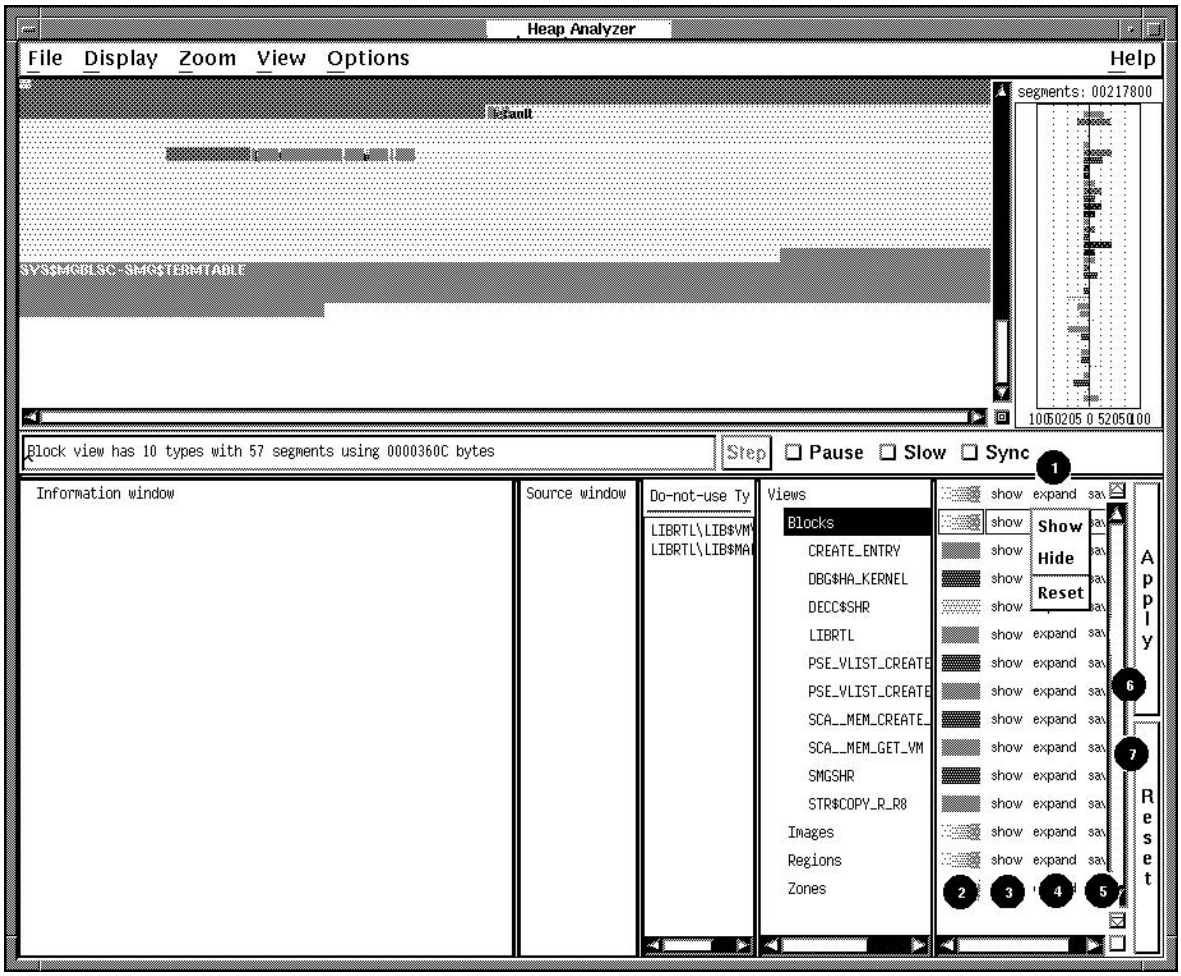
  全セグメント・タイプ，特定のビューだけの全セグメント・タイプ，または個々のセグメント・タイプについての情報を削除 (または保存) するには，「Save」ボタンを選択して設定を「Remove」または「Save」に切り替えてから MB1 をクリックする (または，「Expand」ポップアップ・メニューで適切なメニュー項目を選択する方法もある)。その後「Apply」ボタンをクリックして，変更を実際に適用する。

  「Memory Map」を完全に消去してからもう一度表示する場合に，このオプションを使用する。会話型のコマンドを調べる場合にこのオプションがいかに有効であるかの説明については，第 12.5 節を参照。

選択を取り消すには，「Reset」ボタンをクリックするか，または「Show」，「Expand」，「Save」の各ポップアップ・メニューで「Reset」を選択します。

図 12–11 に，「Show」ポップアップ・メニューを示します。このメニューは，「Views-and-Types」ディスプレイのオプション側で MB3 をクリックすると表示されます。すでに強調表示されている「Blocks」が変更の有効範囲です。この図には，「Views-and-Types」ディスプレイのオプション側で使用できるマウスとメニューのすべての選択項目も示してあります。

図 12–11 ヒープ・アナライザの「Views-and-Types」ディスプレイのオプション



1. MB1 のクリック	「Show」，「Expand」，「Save」の各トグル・ボタンを切り替える。
2.「Color」ポップアップ	個々のまたはグループ化したタイプの表示色を調節する。
3.「Show」ポップアップ	選択したセグメント・タイプの表示を制御する。「Show」と「Hide」の各メニュー項目で表示を復元または抑制できる。選択は「Reset」で取り消される。
4.「Expand」ポップアップ	選択したセグメント・タイプ内のセグメントの表示を制御する。「Expand」と「Collapse」の各メニュー項目で表示を復元または抑制できる。選択は「Reset」で取り消される。
5.「Save」ポップアップ	選択したセグメント・タイプについての情報を表示または格納するヒープ・アナライザの機能を制御する。「Remove」メニュー項目は情報をすべて削除する。「Save」は表示機能を元に戻し，情報を保存する。選択は「Reset」で取り消される。
6.「Apply」ボタン	選択した内容を「Memory Map」ディスプレイに適用する。
7.「Reset」ボタン	選択を取り消す。

12.4 ヒープ・アナライザの終了

ヒープ・アナライザを終了するには，ヒープ・アナライザ画面の「File」メニューで「Exit」を選択します。

12.5 サンプル・セッション

この節では，ヒープ・アナライザの各ウィンドウとメニューから得られる情報をまとめて，ユーザ・アプリケーションにある特定のメモリ・リークを見つける方法の例を示します。

この例では，すでにヒープ・アナライザを起動し，ユーザ・アプリケーションを実行してあると考えます。「Memory Map」ディスプレイをスクロールして戻しながら，ユーザ・アプリケーションが会話型コマンドを呼び出したときに表示されるセグメントに注目します。

12.5.1 会話型コマンドの表示の取り出し

メモリ・リークが生じたのは，会話型コマンド SHOW UNITS を入力したときかもしれません。そこで最初の手順として，「Memory Map」を消去し，このコマンドを再入力してみます。

「Memory Map」を消去して SHOW UNITS コマンドを再入力するには，次の手順に従います。

1. 「Views-and-Types」ディスプレイの「Views」項目の「Remove」オプションをクリックしてから，「Apply」ボタンをクリックする。

   「Memory Map」内の以前の出力がすべて消去される。

2. 「Views」項目の「Save」オプションをクリックしてから，「Apply」ボタンをクリックする。

   以後の「Memory Map」への出力が，すべて保存される。

3. 別の DECterm ウィンドウ内で，ユーザ・アプリケーションのプロンプトに SHOW UNITS コマンドをいくつか入力する。

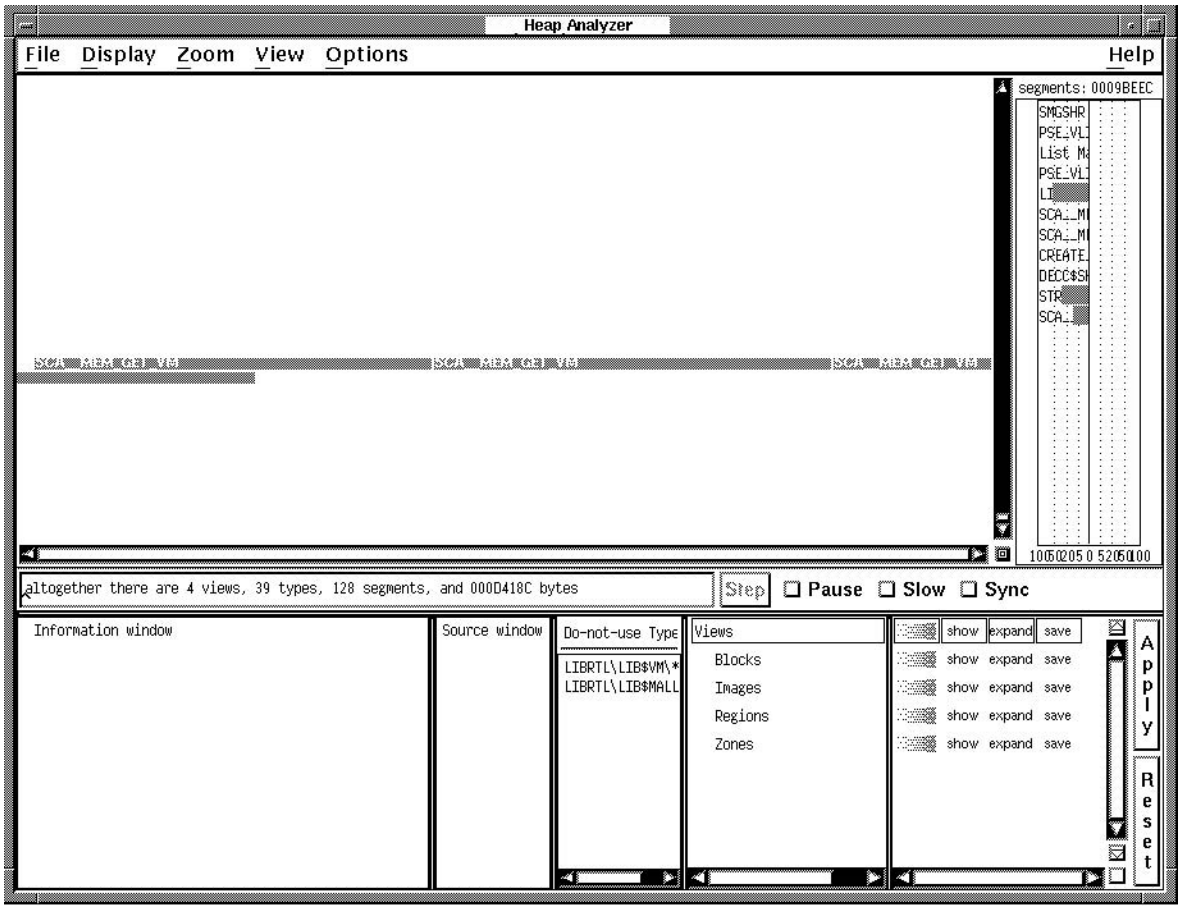
   小さく表示された一連のセグメントは増分されているらしいが，表示が小さすぎて確認できない。

4. 「Zoom」メニューの「Extremely Close」を選択する。

   セグメントのビューが拡大表示される。

それぞれの SCA__MEM_GET_VM セグメントに割り当てられたメモリ空間は，SHOW UNITS コマンドごとに少しずつ増大しています (図 12–12 を参照)。同一サイズのはずの割り当てが増大していることは，メモリ・リークの発生を示しています。

図 12–12 メモリ・リークを示すメモリ割り当ての増分



12.5.2 タイプ設定の変更

各セグメントに対応したセグメント・タイプには，SCA__MEM_GET_VM というラベルが付けられています。これはかなり下位レベルでのメモリ管理ルーチンであり，多数のセグメントがこのルーチンを共有しています。次の手順は，セグメント・タイプを再定義して，抽象化レベルをもっと役に立つものにすることです。できればアプリケーションのルーチン名と対応させます。

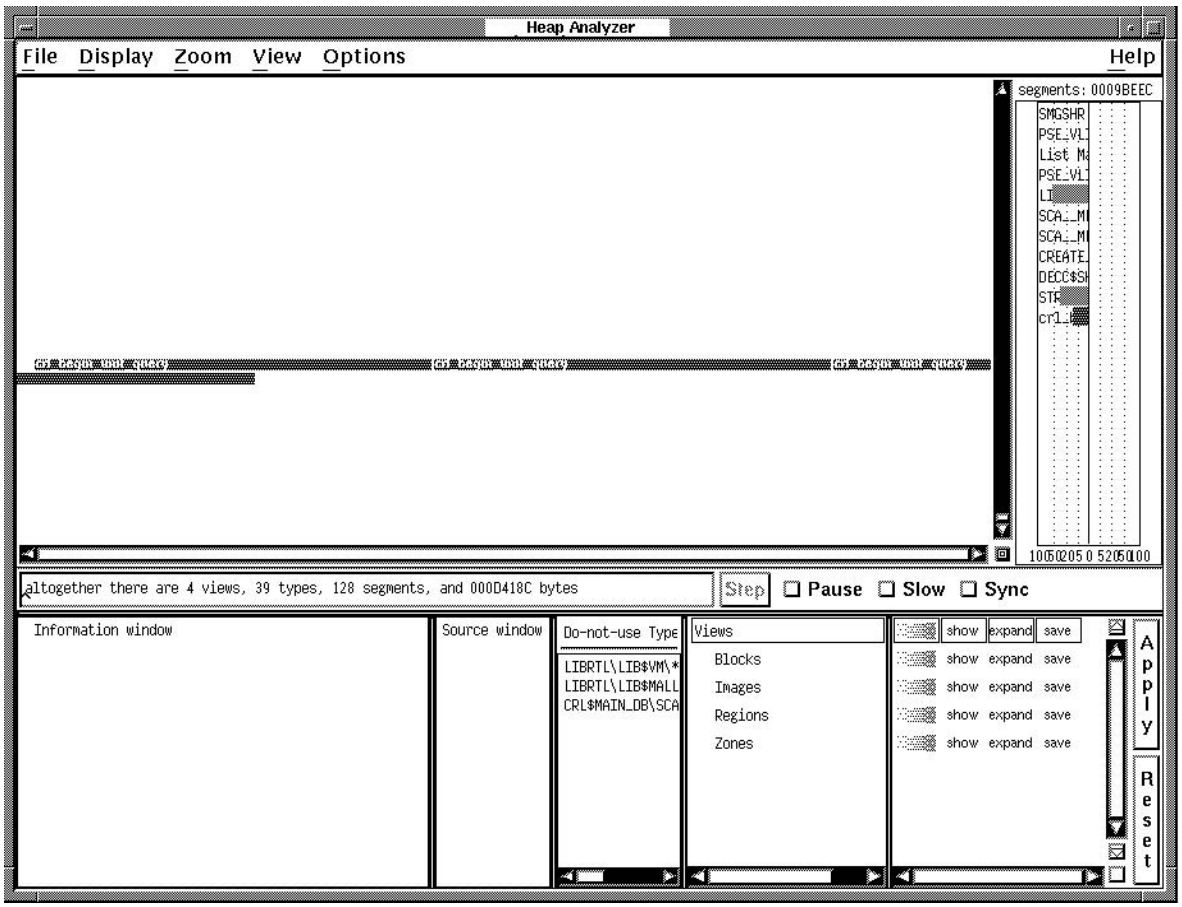
セグメント・タイプを再定義するには，次の手順に従います。

- いずれかのセグメントにマウス・ポインタを置いてから，MB3 をクリックする。

  コンテキスト依存の「Memory Map」ポップアップ・メニューが表示される。
- ポップアップ・メニューで「Do Not Use Type」を選択する。

  セグメントに対応したセグメント・タイプが，SCA__MEM_GET_VM から，有用な "crl_begin_unit_query"に変更される (図 12–13 を参照)。

図 12–13 セグメント・タイプを再定義する「Do-Not-Use Type」メニュー項目



12.5.3 トレースバック情報の表示

セグメントを表示する抽象化レベルを決定したら，次は，セグメントが割り当てられたときの呼び出しスタックの状態を調べます。セグメントごとのトレースバックを調べると，セグメントが作成された時点と理由，およびメモリに問題が生じた理由が分かります。

トレースバック情報を表示するには，次の手順に従います。

1. セグメントにマウス・ポインタを置いてから， MB3 をクリックする。

   コンテキスト依存の「Memory Map」ポップアップ・メニューが表示される。

2. ポップアップ・メニューで「Traceback of Allocation」を選択する。

   セグメントのトレースバック情報が「Information」ウィンドウに表示される。

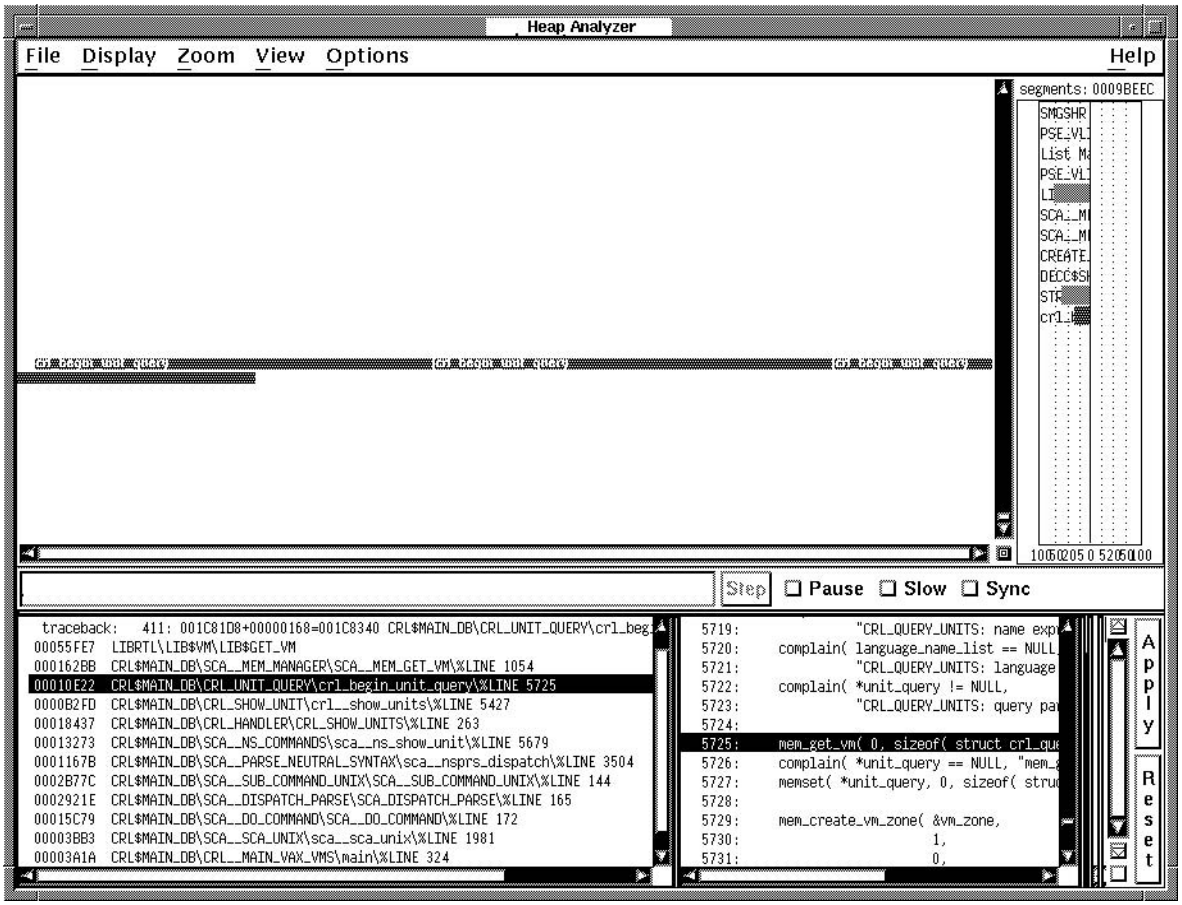
12.5.4 トレースバックとソース・コードとの相互対応

セグメントのトレースバックは， crl_begin_unit_query ルーチンが SHOW UNITS コマンド用の環境を設定していることを示しています。このイベントをさらに詳しく調べるには，イベントに対応したソース・コードを表示させます。

ソース・コードを表示するには， crl_begin_unit_query を参照するトレースバック行を MB1 でダブルクリックします。

「Source」ウィンドウにソース・コードが表示されます。 crl_begin_unit_query を呼び出しスタックに置いたルーチン呼び出しが強調表示されます (図 12–14 を参照)。

図 12–14 トレースバック・エントリのクリックによる，対応したソース・コードの表示

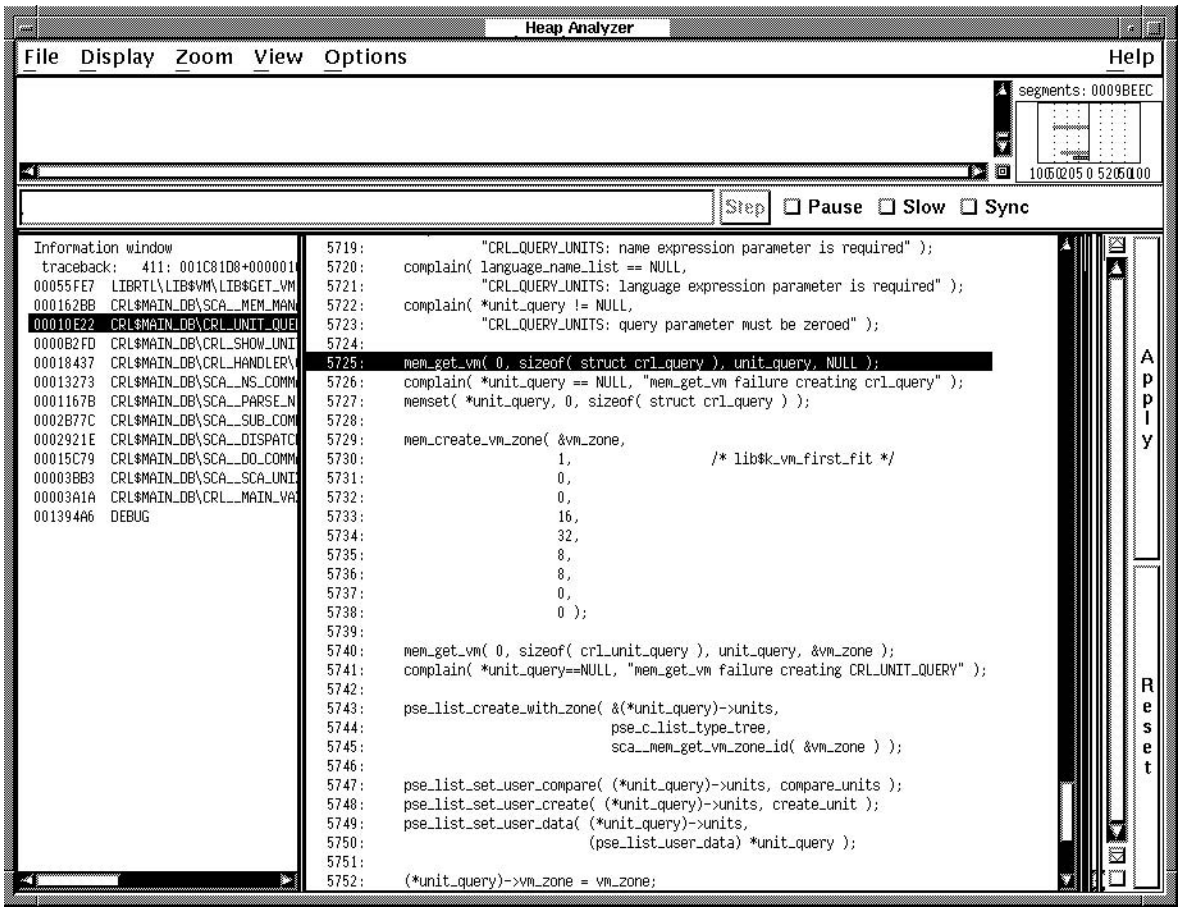


12.5.5 ソース・コードにある割り当てエラーの発見

トレースバック・エントリをアプリケーション・ソース・コードのルーチンと結び付けた後，「Source」ウィンドウを拡大して，ソース・コードの記憶位置で割り当てエラーを探すことができます。

たとえば，図 12–15 では，強調表示されている 5725 行目が unit_query への割り当てを行っています。この割り当ては，別の割り当てが行われる 5740 行目より前で割り当て解除されていません。このコーディング・エラーがメモリ・リークの原因です。

図 12–15 二重の割り当てを示すソース・コード



13

その他の便利な機能

本章では，デバッガが持つ次の便利な機能について説明します。

- デバッガ・コマンド・プロシージャの使用
- デバッグ・セッション用の初期化ファイルの使用
- ログ・ファイルへのデバッグ・セッションの記録
- コマンド，アドレス式，値を表現するシンボルの定義
- ファンクション・キーへのデバッガ・コマンドの割り当て
- コマンド入力のための制御構造の使用
- ユーザ・プログラムにリンクされた任意のルーチンの呼び出し

13.1 デバッガ・コマンド・プロシージャの使用

デバッガ・コマンド・プロシージャは，ファイルに記述された一連のコマンドです。デバッガがコマンド・プロシージャを実行するよう指定して，デバッグ・セッションを再作成したり，前のセッションを続行したり，デバッグ・セッションの間に同じデバッガ・コマンドを何度も入力しなくてもすむようにしたりできます。コマンド・プロシージャにはパラメータを引き渡すことができます。

DCL コマンド・プロシージャの場合と同様に，デバッガ・コマンド・プロシージャを実行するには，ファイル指定の先頭にアットマーク(@)を付けます。@はプロシージャ実行コマンドです。

デバッガ・コマンド・プロシージャは，デバッガ初期化ファイル (第 13.2 節を参照) に指定されているような標準的な設定用デバッガ・コマンドを数多く定期的に実行する場合に特に便利です。また，デバッガ・ログ・ファイルをコマンド・プロシージャとして使用することもできます (第 13.3 節を参照)。

13.1.1 基本的な規則

次に示すのは，デバッガ・コマンド・プロシージャ BREAK7.COM の例です。

```
! ***** デバッガ・コマンド・プロシージャBREAK7.COM *****
SET BREAK/AFTER:3 %LINE 120 DO (EXAMINE K,N,J,X(K); GO)
SET BREAK/AFTER:3 %LINE 160 DO (EXAMINE K,N,J,X(K),S; GO)
SET BREAK %LINE 90
```

このコマンド・プロシージャをプロシージャ実行コマンド(@)で実行すると，プロシージャ内に記述されたコマンドがその順番に実行されます。

コマンド・プロシージャへのコマンド入力の規則については，デバッガのオンライン・ヘルプを参照してください (HELP Command_Formatと入力します)。

コマンド・プロシージャには，パラメータを引き渡すことができます。パラメータ引き渡しの規則については，第 13.1.2 項を参照してください。

プロシージャ実行 (@) コマンドは，他のデバッガ・コマンドと同様に入力することができます。すなわち，端末から直接入力したり，別のコマンド・プロシージャ内から入力したり， SET BREAK などのコマンドの DO 句から入力したり，画面表示定義の DO 句から入力したりすることができます。

プロシージャ実行 (@) コマンドに指定するファイル指定が完全ファイル指定ではない場合，デバッガは SYS$DISK:[ ]DEBUG.COM をコマンド・プロシージャの省略時のファイル指定とみなします。たとえば，現在の省略時のディレクトリにあるコマンド・プロシージャ BREAK7.COM を実行するためには，次のコマンド行を入力します。

```
DBG> @BREAK7
```

SET ATSIGN コマンドは，省略時のファイル指定である SYS$DISK:[ ]DEBUG.COM の一部またはすべてのフィールドの変更を可能にします。 SHOW ATSIGN コマンドは，コマンド・プロシージャの省略時のファイル指定を表示します。

省略時の設定では，コマンド・プロシージャから読み込まれたコマンドはエコーバックされません。 SET OUTPUT VERIFY コマンドを入力すると，コマンド・プロシージャから読み込まれたすべてのコマンドが， DBG$OUTPUT で指定された現在の出力装置にエコーバックされます。省略時の出力装置は SYS$OUTPUT です。コマンド・プロシージャから読み込まれたコマンドがエコーバックされるかどうかを明らかにするには， SHOW OUTPUT コマンドを使用します。

コマンド・プロシージャ内のコマンドを実行した結果，警告かそれ以上の重大な診断メッセージが表示された場合，そのコマンドは強制終了されます。ただしコマンド・プロシージャの実行は次のコマンド行から続けられます。

13.1.2 コマンド・プロシージャへのパラメータの引き渡し

DCL コマンド・プロシージャの場合と同様に，デバッガ・コマンド・プロシージャにもパラメータを引き渡すことができます。ただし，いくつかの点でその方法が異なります。

ここで説明する規則に従って，ユーザはデバッガ・コマンド・プロシージャに対して必要なだけパラメータを引き渡すことができます。パラメータには，アドレス式，コマンド，または現在使用中の言語で記述された値式を指定できます。コマンド文字列は二重引用符(")で囲み，各パラメータはコンマ(,)で区切らなければなりません。

パラメータが引き渡される側のデバッガ・コマンド・プロシージャには，引き渡された実パラメータと，コマンド・プロシージャ内で宣言された仮パラメータ (シンボル) とを結び付ける DECLARE コマンド行が必要です。

DECLARE コマンドは，コマンド・プロシージャ内でのみ有効です。 DECLARE コマンドの構文は次のとおりです。

```
DECLARE p-name:p-kind[, p-name:p-kind[, . . . ]]
```

それぞれの*p-name:p-kind*の組み合わせは，仮パラメータ (*p-name*) をパラメータの種類 (*p-kind*) と関連づけています。有効な*p-kind*キーワードを次に示します。

ADDRESS	実パラメータをアドレス式として解釈する
COMMAND	実パラメータをコマンドとして解釈する
VALUE	実パラメータを現在使用中の言語で記述された値式として解釈する

次の例は，パラメータがコマンド・プロシージャに引き渡されたとき，どのように処理されるかを示しています。コマンド・プロシージャ EXAM.COM の中の DECLARE K:ADDRESS コマンドは，仮パラメータ K を宣言しています。EXAM.COM に引き渡される実パラメータはアドレス式として解釈されます。EXAMINE K コマンドは，そのアドレス式の値を表示します。 SET OUTPUT VERIFY コマンドは，コマンドがデバッガに読み込まれたとき，それらをエコーバックします。

```
! ***** デバッガ・コマンド・プロシージャEXAM.COM *****
SET OUTPUT VERIFY
DECLARE DBG:ADDRESS
EXAMINE DBG
```

次のコマンド行は， EXAM.COM を実行します。このとき実パラメータ ARR24 を引き渡します。 EXAM.COM 内で ARR24 はアドレス式 (この場合は配列変数) として解釈されます。

```
DBG> @EXAM ARR24
%DEBUG-I-VERIFYIC, entering command procedure EXAM
 DECLARE DBG:ADDRESS
 EXAMINE DBG
PROG_8\ARR24
    (1):        Mark A. Hopper
    (2):        Rudy B. Hopper
    (3):        Tim B. Hopper
    (4):        Don C. Hopper
    (5):        Mary D. Hopper
    (6):        Jeff D. Hopper
    (7):        Nancy G. Hopper
    (8):        Barbara H. Hopper
    (9):        Lon H. Hopper
   (10):        Dave H. Hopper
   (11):        Andy J. Hopper
   (12):        Will K. Hopper
   (13):        Art L. Hopper
   (14):        Jack M. Hopper
   (15):        Karen M. Hopper
   (16):        Tracy M. Hopper
   (17):        Wanfang M. Hopper
   (18):        Jeff N. Hopper
   (19):        Nancy O. Hopper
   (20):        Mike R. Hopper
   (21):        Rick T. Hopper
   (22):        Dave W. Hopper
   (23):        Jim W. Hopper
   (24):        Robert Z. Hopper
%DEBUG-I-VERIFYIC, exiting command procedure EXAM
DBG>
```

DECLARE コマンドで指定されたそれぞれの*p-name*:*p-kind*の組み合わせは，1つのパラメータを結び付けます。たとえば1つのコマンド・プロシージャに5つのパラメータを引き渡す場合，5つの対応する*p-name*:*p-kind*の組み合わせが必要になります。これらの組み合わせは常に指定した順番に処理されます。

たとえば，次のコマンド・プロシージャ EXAM_GO.COM は，アドレス式 (L) とコマンド文字列 (M) の 2 つのパラメータを受け取ります。その後，アドレス式が調べられ，コマンドが実行されます。

```
! ***** デバッガ・コマンド・プロシージャEXAM_GO.COM *****
DECLARE L:ADDRESS, M:COMMAND
EXAMINE L; M
```

次の例は，検査対象の変数 X，実行対象のコマンド@DUMP.COM を渡す EXAM_GO.COM の実行方法を示しています。

```
DBG> @EXAM_GO X, "@DUMP"
```

組み込みシンボル%PARCNT は，コマンド・プロシージャ内だけで使用できるものであり，コマンド・プロシージャへさまざまな個数のパラメータを引き渡せるようにします。%PARCNT の値は，コマンド・プロシージャへ引き渡す実パラメータの個数を示します。

次の例では，組み込みシンボル%PARCNT が使用されています。コマンド・プロシージャ VAR.DBG には，次の行が含まれています。

```
! ***** デバッガ・コマンド・プロシージャVAR.DBG *****
SET OUTPUT VERIFY
! 引き渡されるパラメータの個数を表示する。
EVALUATE %PARCNT
! 引き渡されたパラメータをすべて結合して値を取得するまでループする。
FOR I = 1 TO %PARCNT DO (DECLARE X:VALUE; EVALUATE X)
```

次のコマンド行は， VAR.DBG にパラメータ 12， 37， 45 を引き渡して実行します。

```
DBG> @VAR.DBG 12,37,45
%DEBUG-I-VERIFYIC, entering command procedure VAR.DBG
! 引き渡されるパラメータの個数を表示する。
EVALUATE %PARCNT
3
! 引き渡されたパラメータをすべて結合して，
! 値を取得するまでループする。
FOR I = 1 TO %PARCNT DO (DECLARE X:VALUE; EVALUATE X)
12
37
45
%DEBUG-I-VERIFYIC, exiting command procedure VAR.DBG
DBG>
```

VAR.DBG が実行されるとき，%PARCNT の値は 3 です。したがって， VAR.DBG 内の FOR ループは 3 回繰り返されます。 FOR ループでは， DECLARE コマンドが 3 つの各実パラメータ (12 が最初) を新しい宣言 X に結び付けます。それぞれの実パラメータは，現在使用中の言語の値式として解釈され， EVALUATE X コマンドがその値を表示します。

13.2 デバッガ初期化ファイルの使用

デバッガ初期化ファイルはコマンド・プロシージャであり，論理名 DBG$INIT が付けられています。このファイルはデバッガの起動時に自動的に実行されます。デバッガを起動するたびにファイル内に記述されたコマンドが自動的に実行されます。

初期化ファイルには，デバッグ環境を整えるため，またはユーザ・プログラムが毎回の実行時にあらかじめ決められた方法で実行されるよう制御するために，デバッグ・セッションを開始するときに必ず入力しなければならないコマンド行が含まれています。

たとえば，ファイル DEBUG_START4.COM に次のコマンドが含まれているとします。

```
! ***** デバッガ初期化ファイルDEBUG_START4.COM *****
! デバッグ・セッションを省略時のログ・ファイル(SYS$DISK:[]DEBUG.LOG)に記
録する。
SET OUTPUT LOG
!
! コマンド・プロシージャから読み込まれたコマンドをエコーバックする。
SET OUTPUT VERIFY
!
! ソース・ファイルが現在の省略時ディレクトリにない場合は[SMITH.SHARE]を
使用する。
SET SOURCE [],[SMITH.SHARE]
!
! 画面モードを起動する。
SET MODE SCREEN
!
! シンボルSBをSET BREAKコマンドとして定義する。
DEFINE/COMMAND SB = "SET BREAK"
!
! SHOW MODULE *コマンドをKP7に割り当てる。
DEFINE/KEY/TERMINATE KP7 "SHOW MODULE *"
```

このファイルをデバッガ初期化ファイルにするには， DCL の DEFINE コマンドを使用します。次に例を示します。

```
$ DEFINE DBG$INIT WORK:[JONES.DBGCOMFILES]DEBUG_START4.COM
```

13.3 ログ・ファイルへのデバッグ・セッションの記録

デバッガ・ログ・ファイルは，デバッグ・セッションの履歴を記録します。デバッグ・セッションの間に，入力された各コマンドとその結果のデバッガの出力がファイルに保存されます。次にデバッガ・ログ・ファイルの例を示します。

```
SHOW OUTPUT
!noverify, terminal, noscreen_log, logging to DSK2:[JONES.P7]DEBUG.LOG;1
SET STEP NOSOURCE
SET TRACE %LINE 30
SET BREAK %LINE 60
SHOW TRACE
!tracepoint at PROG4\%LINE 30
GO
!trace at PROG4\%LINE 30
!break at PROG4\%LINE 60
     .
     .
     .
```

DBG>プロンプトは記録されず，デバッガの出力内容は感嘆符が付けられてコメントになるので，ファイルは修正なしでデバッガ・コマンド・プロシージャとして使用することができます。したがって，長いデバッグ・セッションに割り込みがかかったとき，ログ・ファイルを他のデバッガ・コマンド・プロシージャと同じように実行することができます。ログ・ファイルを実行すると，前回中断したところまでデバッグ・セッションが復元されます。

デバッガ・ログ・ファイルを作成するには， SET OUTPUT LOG コマンドを使用します。省略時には，デバッガはログを SYS$DISK:[ ]DEBUG.LOG に書き込みます。デバッガ・ログ・ファイルに名前を付けるには， SET LOG コマンドを使用します。省略時のファイル指定のどのフィールドでも上書きすることができます。たとえば，次のコマンドを入力すると，その後デバッガはセッションをファイル[JONES.WORK2]MONITOR.LOG に記録します。

```
DBG> SET LOG [JONES.WORK2]MONITOR
DBG> SET OUTPUT LOG
```

SET OUTPUT LOG コマンドをデバッガ初期化ファイルに入力する場合もあります。第 13.2 節を参照してください。

SHOW LOG コマンドは，デバッガがログ・ファイルに書き込みを行っているかどうかを報告し，現在のログ・ファイルを示します。 SHOW OUTPUT コマンドは，現在の出力オプションをすべて示します。

画面モードでデバッグしているとき， SET OUTPUT SCREEN_LOG コマンドを使用して画面の内容を更新される様子そのままに記録することができます。このコマンドを使用するには，デバッグ・セッションのログがすでに記録中になっていなければなりません。すなわち， SET OUTPUT SCREEN_LOG コマンドは， SET OUTPUT LOG コマンドを入力したあとでだけ有効になります。 SET OUTPUT SCREEN_LOG コマンドで画面情報を保存すると大きなログ・ファイルが作成されるため，長いデバッグ・セッションで使用することは望ましくありません。画面モードの情報を保存するその他の方法については， SAVE コマンドおよび EXTRACT コマンドの説明を参照してください。

ログ・ファイルをコマンド・プロシージャとして使用するときは，まず SET OUTPUT VERIFY コマンドを入力して，読み込まれたデバッガ・コマンドをエコーバックするようにしなければなりません。

13.4 コマンド，アドレス式，値の各シンボルの定義

DEFINE コマンドは，長かったり，繰り返し使用されたりするコマンド・シーケンスやアドレス式のシンボルを作成したり，言語式の値をシンボルに格納したりするために使用します。

定義したいシンボルの種類は， DEFINE コマンドに使用するコマンド修飾子 (/COMMAND，/ADDRESS，または/VALUE) で指定します。省略時の修飾子は/ADDRESS です。いくつかの DEFINE コマンドで同じ修飾子を使用する場合は，最初に SET DEFINE コマンドを使用して省略時の修飾子を新しく設定することができます。たとえば， SET DEFINE COMMAND は， DEFINE コマンドを DEFINE/COMMAND として動作するように設定します。 SHOW DEFINE コマンドは，現在有効な省略時の修飾子を示します。

SHOW SYMBOL/DEFINED コマンドは， DEFINE コマンドで定義したシンボルを示すために使用します。 SHOW SYMBOL コマンドに/DEFINED 修飾子を付けないと，ユーザ・プログラム内で定義されたルーチン名や変数名などのシンボルだけを示すので注意してください。

DELETE コマンドは， DEFINE コマンドで作成されたシンボルの定義を削除するために使用します。

コマンド・プロシージャ内でシンボルを定義する場合，/LOCAL 修飾子を使用してシンボルの定義をそのコマンド・プロシージャ内だけに制限することができます。

13.4.1 コマンドのシンボルの定義

DEFINE/COMMAND コマンドは， 1つまたは複数のコマンド文字列をより短いシンボルに等しいと定義するために使用します。基本的な構文を次の例に示します。

```
DBG> DEFINE/COMMAND SB = "SET BREAK"
DBG> SB PARSER
```

この例では， DEFINE/COMMAND コマンドはシンボル SB が文字列 SET BREAK に等しいと定義します。コマンド文字列を区切るために二重引用符が使用されていることに注意してください。コマンド行 SB PARSER が実行されると，デバッガはシンボル SB を文字列 SET BREAK に置き換え， SET BREAK コマンドを実行します。

次の例では， DEFINE/COMMAND コマンドは，シンボル BT を SHOW BREAK コマンドとそれに続く SHOW TRACE コマンドからなる文字列に等しいと定義しています。複数のコマンドを指定する文字列では，コマンド間にセミコロンを挿入して区切ります。

```
DBG> DEFINE/COMMAND BT = "SHOW BREAK;SHOW TRACE"
```

SHOW SYMBOL/DEFINED コマンドは，シンボル BT を次のように示します。

```
DBG> SHOW SYM/DEFINED BT
defined BT
    bound to: "SHOW BREAK;SHOW TRACE"
    was defined /command
DBG>
```

複雑なコマンドを定義するために，コマンド・プロシージャにパラメータを付けて使用しなければならない場合があります。コマンド・プロシージャへのパラメータの引き渡しについては，第 13.1.2 項を参照してください。次に例を示します。

```
DBG> DEFINE/COMMAND DUMP =  "@DUMP_PROG2.COM"
```

13.4.2 アドレス式のシンボルの定義

DEFINE/ADDRESS コマンドは，アドレス式をシンボルと等しいと定義するために使用します。/ADDRESS 修飾子は DEFINE コマンドの省略時の修飾子ですが，ここで使用する例では，強調のために指定されています。

次の例では，シンボル B1 は行 378 のアドレスに等しいと定義されています。次の SET BREAK B1 コマンドは，行 378 にブレークポイントを設定します。

```
DBG> DEFINE/ADDRESS B1 = %LINE 378
DBG> SET BREAK B1
```

DEFINE/ADDRESS コマンドは，何度も定義される変数名やルーチン名を参照するために，繰り返し長いパス名を指定しなければならないときに使用すると便利です。次の例では，シンボル UX がパス名 SCREEN_IO\UPDATE\X に等しいと定義されています。短縮された形のコマンド行 EXAMINE UX を次に使用して，モジュール SCREEN_IO のルーチン UPDATE 内での X の値を取得しています。

```
DBG> DEFINE UX = SCREEN_IO\UPDATE\X
DBG> EXAMINE UX
```

13.4.3 値のシンボルの定義

DEFINE/VALUE コマンドは，言語式の現在の値とシンボルを等しいと定義するために使用します。現在の値とは， DEFINE/VALUE コマンドが入力された時点の値のことを指します。

次に， DEFINE/VALUE コマンドがルーチンの呼び出しの回数を数えるために使用されている例を示します。

```
DBG> DEFINE/VALUE COUNT = 0
DBG> SET TRACE/SILENT ROUT DO (DEFINE/VALUE COUNT = COUNT + 1)
DBG> GO
   .
   .
   .
DBG> EVALUATE COUNT
14
DBG>
```

この例では，最初の DEFINE/VALUE コマンドがシンボル COUNT の値を 0 に初期化しています。 SET TRACE コマンドは，ルーチン ROUT にサイレント・トレースポイントを設定し， DO 句を通して ROUT が呼び出されるたびに COUNT の値を 1 ずつ増分します。実行が再開され中断したとき， EVALUATE コマンドは COUNT の現在の値 (ROUT が呼び出された回数) を取得します。

13.5 ファンクション・キーへのコマンドの割り当て

よく使用するコマンドを簡単に入力できるようにするために，キーパッド上のファンクション・キーにはデバッガの起動字に設定される機能があらかじめ定義されています。これらの定義済みの機能については，付録 A を参照してください。キーパッド・キーの機能は，各ユーザのニーズに応じて変更できます。 VT200 シリーズまたは VT300 シリーズの端末やワークステーションを使用している場合は， LK201 キーボードのその他のファンクション・キーにコマンドを割り当てることができます。

デバッガ・コマンドの DEFINE/KEY， SHOW KEY，および DELETE/KEY は，それぞれキー定義の割り当て，表示，削除を行います。これらの機能を使用する前に，SET MODE KEYPAD コマンドでキーパッド・モードが有効になっていなければなりません。省略時の設定ではキーパッド・モードは有効になっています。キーパッド・モードでは，キーパッド・キーの定義済みの機能を使用することもできます。

キーパッド・キーをデバッガ・コマンドの入力ではなく数字の入力用に使用するときは， SET MODE NOKEYPAD コマンドを使用します。

13.5.1 基本的な規則

デバッガの DEFINE/KEY コマンドは， DCL の DEFINE/KEY コマンドに類似しており，文字列をファンクション・キーへ割り当てます。次の例では， DEFINE/KEY コマンドは SHOW MODULE *コマンドを入力し実行する機能を KP7 (キーパッド・キーの 7) に定義しています。

```
DBG> DEFINE/KEY/TERMINATE KP7 "SHOW MODULE *"
%DEBUG-I-DEFKEY, DEFAULT key KP7 has been defined
DBG>
```

DEFINE/KEY コマンド， SHOW KEY コマンド， DELETE/KEY コマンドに対しては，正しいキー名 (KP7 など) を使用しなければなりません。 VT52 および VT100 シリーズの端末と LK201 キーボードでこれらのコマンドに対して使用できる正しいキー名については， DEFINE/KEY コマンドの説明を参照してください。

前の例では，/TERMINATE 修飾子は KP7 を押すとコマンドが実行されるようにしています。このとき， KP7 を押したあとに Return キーを押す必要はありません。

各定義がそれぞれ別の状態に関連づけられているかぎり，同じ 1 つのファンクション・キーに対して，いくつでも定義を割り当てることができます。定義済みの状態 (DEFAULT， GOLD， BLUE など) については付録 A を参照してください。前の例では， KP7 は DEFAULT 状態に定義されていることをメッセージが示しています。これは省略時のキー状態です。

キー定義をデバッガ初期化ファイル (第 13.2 節を参照) に入力し，デバッガを起動したときすぐにそれらの定義を利用できるようにすることができます。

現在の状態のキー定義を表示するには， SHOW KEY コマンドを入力します。次に例を示します。

```
DBG> SHOW KEY KP7
DEFAULT keypad definitions:
  KP7 = "SHOW MODULE *" (echo,terminate,nolock)
DBG>
```

現在の状態以外の状態のキー定義を表示するには， SHOW KEY コマンドを入力するときに/STATE 修飾子を付けてその状態を指定します。現在の状態のすべてのキー定義を表示するには， SHOW KEY/ALL コマンドを入力します。

キー定義を削除するには， DELETE/KEY コマンドを使用します。現在の状態以外の状態のキー定義を削除する場合には，/STATE 修飾子を付けてその状態を指定します。次に例を示します。

```
DBG> DELETE/KEY/STATE=GOLD KP7
%DEBUG-I-DELKEY, GOLD key KP7 has been deleted
DBG>
```

13.5.2 より高度な方法

ここでは，キー定義のより高度な方法について説明します。特に，状態キーの使用に関して説明します。

次のコマンド行は，コマンドとしては不完全な文字列"SET BREAK %LINE"を KP9 の BLUE 状態に割り当てます。

```
DBG> DEFINE/KEY/IF_STATE=BLUE KP9 "SET BREAK %LINE"
```

定義済みの DEFAULT キー状態は，省略時に設定されます。定義済みの BLUE キー状態は， PF4 キーを押すことによって設定されます。前の例 (SET BREAK %LINE . . .) で割り当てられたコマンド行を入力するには， PF4 を押して KP9 を押し，行番号を入力し，最後に Return キーを押してコマンド行を終了させ処理します。

SET KEY コマンドは，キー定義の省略時の状態を変更します。たとえば， SET KEY/STATE=BLUE コマンドの入力後，前の例のコマンド行を入力する場合， PF4 を押す必要はありません。また， SHOW KEY コマンドは省略時に BLUE 状態のキー定義を表示するようになり， DELETE/KEY コマンドは省略時に BLUE 状態のキー定義を削除するようになります。

その他のキー状態を作成することができます。次に例を示します。

```
DBG> SET KEY/STATE=DEFAULT
DBG> DEFINE/KEY/SET_STATE=RED/LOCK_STATE F12 ""
```

この例では， SET KEY コマンドは現在の状態として DEFAULT を設定します。 DEFINE/KEY コマンドは， F12(LK201 キーボード) を状態キーとしています。その結果， DEFAULT 状態で F12 を押すと現在の状態は RED になります。キー定義は終了せず，他に何も行いません。空文字列が F12 に割り当てられます。 F12 を押したあと， RED 状態に関連づけられた定義を持つキーを押すと， RED のコマンドを入力することができます。

13.6 コマンド入力のための制御構造の使用

FOR， IF， REPEAT， WHILE の各コマンドを使用すると，デバッガ・コマンドを入力するためのループ構造や条件付き構造を作成することができます。 FOR, REPEAT， WHILE のループを終了するには，対応するコマンドの EXITLOOP を使用します。以降の項ではこれらのコマンドについて説明します。

言語式の評価についての詳しい説明は，第 4.1.6 項および第 14.3.2.2 項を参照してください。

13.6.1 FOR コマンド

FOR コマンドは，指定された回数だけ変数を増分する間，一連のコマンドを実行します。 FOR コマンドの構文は次のとおりです。

```
FOR name=expression1 TO expression2 [BY expression3] DO(command[; . . . ])
```

たとえば，次のコマンド行は配列の最初の 10 個の要素をゼロに初期化するためのループを設定します。

```
DBG> FOR I = 1 TO 10 DO (DEPOSIT A(I) = 0)
```

13.6.2 IF コマンド

IF コマンドは，言語式 (論理式) が真と評価されたとき一連のコマンドを実行します。 IF コマンドの構文は次のとおりです。

```
IF boolean-expression THEN (command[; . . . ]) [ELSE (command[; . . . ])]
```

次の FORTRAN の例では， X1 が–9.9 に等しくないとき EXAMINE X2 コマンドを実行し，等しいときに EXAMINE Y1 コマンドを実行する，という条件を設定しています。

```
DBG> IF X1 .NE. -9.9 THEN (EXAMINE X2) ELSE (EXAMINE Y1)
```

次の Pascal の例では， FOR ループと条件のテストを結合しています。 X1 が–9.9 に等しくないとき， STEP コマンドが実行されます。テストは 4 回行われます。

```
DBG> FOR COUNT = 1 TO 4 DO (IF X1 <> -9.9 THEN (STEP))
```

13.6.3 REPEAT コマンド

REPEAT コマンドは，指定された回数だけ一連のコマンドを実行します。 REPEAT コマンドの構文は次のとおりです。

```
REPEAT language-expression DO (command[; . . . ])
```

たとえば，次のコマンド行は連続する 2 つのコマンド (EXAMINE Y と STEP) を 10 回実行するループを設定します。

```
DBG> REPEAT 10 DO (EXAMINE Y; STEP)
```

13.6.4 WHILE コマンド

WHILE コマンドは，指定した言語式 (論理式) が真と評価される間，一連のコマンドを実行します。 WHILE コマンドの構文は次のとおりです。

```
WHILE boolean-expression DO (command[; . . . ])
```

次の Pascal の例は， X1 と X2 を繰り返しテストし， X2 が X1 より小さければ， 2 つのコマンド EXAMINE X2 および STEP を実行するループを設定します。

```
DBG> WHILE X2 < X1 DO (EX X2;STEP)
```

13.6.5 EXITLOOP コマンド

EXITLOOP コマンドは，1 つまたは複数の FOR，REPEAT，または WHILE のループを終了します。EXITLOOP コマンドの構文は次のとおりです。

```
EXITLOOP [integer]
```

整数*n*は，ネストしたループをいくつ終了するかを指定します。

次の Pascal の例では，繰り返しのたびに STEP コマンドを実行する永久ループを設定しています。各ステップの実行後，X の値がテストされます。X が 3 より大きければ，EXITLOOP コマンドによりループが終了します。

```
DBG> WHILE TRUE DO (STEP; IF X > 3 THEN EXITLOOP)
```

13.7 プログラムの実行から独立したルーチンの呼び出し

CALL コマンドを使用して，ユーザ・プログラムの通常の実行からは独立して，ルーチンを実行できます。このコマンドは，ユーザ・プログラムを実行する 4 つのデバッガ・コマンドのうちの 1 つです。その他のコマンドは GO，STEP，および EXIT です。

CALL コマンドは，プログラムが実際にそのルーチンを呼び出しているかどうかに関係なく，ルーチンを実行します。呼び出されるルーチンはユーザ・プログラムにリンクされていることが必要です。したがって，CALL コマンドを使用すればどんな目的のためにでもルーチンを実行することができます。たとえば，プログラム実行のコンテキストに無関係にルーチンをデバッグしたり，実行時ライブラリ・プロシージャを呼び出したり，デバッグ情報をダンプするルーチンを呼び出したりするために使用できます。

関連のないルーチンは，転送アドレスを持つ仮のメイン・プログラムにリンクし，それらを CALL コマンドを使用して呼び出すことによって，デバッグすることができます。

次の例は，必要なコードをプログラム内に記述せずに，プロセス統計を表示するために CALL コマンドを使用する方法を示しています。この例は，実行時ライブラリ・ルーチンのうち，タイマを初期化するルーチン (LIB$INIT_TIMER) と，経過時間および種々の統計情報を表示するルーチン (LIB$SHOW_TIMER) への呼び出しから構成されています。デバッガの存在がタイミングやカウントに影響を与えることに注意してください。

```
DBG> SET MODULE SHARE$LIBRTL   1
DBG> CALL LIB$INIT_TIMER   2
value returned is 1   3
DBG> [ enter various debugger commands ]
   .
   .
   .
DBG> CALL LIB$SHOW_TIMER   4
 ELAPSED: 0 00:00:21.65  CPU: 0:14:00.21  BUFIO: 16  DIRIO: 0  FAULTS: 3
value returned is 1
DBG>
```

次の番号は，上記の例の番号に対応しています。

1 ルーチン LIB$INIT_TIMER および LIB$SHOW_TIMER は，共用可能なイメージ LIBRTL 内にあります。そのイメージは，それ自身のモジュールの設定によって設定されなければなりません。これは，そのユニバーサル・シンボルだけがデバッグ・セッションの間にアクセス可能だからです。第 5.4.2.3 項を参照してください。

2 この CALL コマンドはルーチン LIB$INIT_TIMER を実行します。

3 value returned のメッセージは， CALL コマンドの実行後， R0 レジスタに戻された値を示しています。

慣例によって，呼び出されたルーチンの実行後，レジスタ R0 には関数の戻り値 (ルーチンが関数である場合)，またはプロシージャ終了状態 (ルーチンが状態値を戻すプロシージャである場合) が入ります。呼び出されたプロシージャが状態値も関数値も戻さない場合， R0 の値は意味を持ちませんので， value returned のメッセージは無視してかまいません。

4 この CALL コマンドはルーチン LIB$SHOW_TIMER を実行します。

次の例は，メモリの統計情報を表示するために， LIBRTL に存在する LIB$SHOW_VM を呼び出す方法を示しています。やはりデバッガの存在がカウントに影響を与えることに注意してください。

```
DBG> SET MODULE SHARE$LIBRTL
DBG> CALL LIB$SHOW_VM
 1785 calls to LIB$GET_VM, 284 calls to LIB$FREE_VM,
 122216 bytes still allocated value returned is 1
DBG>
```

CALL コマンドを実行するときルーチンにパラメータを引き渡すことができます。詳しい説明および例については， CALL コマンドの項を参照してください。

14
特殊なデバッグ

本章では，特殊な場合のデバッグ方法について説明します。

- 最適化されたコード
- 画面用プログラム
- 複数言語プログラム
- スタックの破損
- 例外ハンドラと条件ハンドラ
- 終了ハンドラ
- AST ドライブ式プログラム
- 変換されたイメージ

14.1 最適化されたコードのデバッグ

省略時の設定では，プログラムの実行速度を向上させるためにコードを最適化するコンパイラが多くあります。最適化は，実行時に一度だけ評価されればすむよう不変式が DO ループから除かれる，メモリ記憶位置のいくつかがプログラム内の異なる場所で異なる変数に割り当てられる，いくつかの変数がデバッグ中にアクセスしなくてすむように削除される，などの方法で行われます。

最終的には，デバッグ時に実行されているコードは，画面モードのソース・ディスプレイ (第 7.4.1 項を参照) で表示されるソース・コードやソース・リスト・ファイルに記述されているソース・コードとは一致しないことがあります。

最適化されたコードをデバッグする際の問題を回避するために，多くのコンパイラではコンパイル時に/NOOPTIMIZE，またはそれに等しいコマンド修飾子を指定することができます。この修飾子を指定すると，コンパイラによる最適化がほとんど抑止され，それによって最適化のために発生するソース・コードと実行可能なコードとの相違を減らすことができます。

このオプションが利用できない場合，または最適化されているコードのデバッグが明らかに必要な場合は，本章を参照してください。本章では，最適化されたコードのデバッグ方法と，最適化されたコードの典型的な例を使用して混乱の潜在的な原因を示します。また，最適化されたコードをデバッグするときに，このような混乱を減らす目的で使用する機能についても説明します。

この機能を十分活かした上で，最適化されたコードのデバッグ効率を向上させるためには，使用している言語コンパイラの最新版を入手しなければなりません。必要なコンパイラのバージョンについては，コンパイラのリリース・ノートやその他のマニュアルなどを参照してください。

また最適化されたコードをデバッグするためには，イメージのサイズの増加に対応するため，通常の必要ディスク領域に比べて，約 1/3 の領域が余分に必要になります。

最適化されたコードをデバッグする場合，定義済みディスプレイの INST などの画面モード機械語命令ディスプレイを使用して，プログラムのデコード済み命令ストリーム (第 7.4.4 項を参照) を表示します。機械語命令ディスプレイは，実際に実行されているコードを表示します。

画面モードでは， KP7 を押すと SRC 表示と INST 表示が比較しやすいように並べて表示されます。または，コンパイラが生成した機械語コード・リストを調べることもできます。

さらに，プログラムを命令レベルで実行し，命令を検査するには，第 4.3 節で説明されている方法を使用してください。

これらの方法を使用すると，実行可能なコード・レベルで何が起きているかを明確にし，ソース・ディスプレイとプログラム動作の相違を解決することができます。

14.1.1 削除された変数

コンパイラは，実行中のさまざまな時点で変数を永久にまたは一時的に削除することによってコードを最適化することがあります。たとえば，最適化によってアクセスできなくなった変数 X を検査しようとすると，デバッガは次のメッセージのいずれかを表示します。

```
%DEBUG-W-UNALLOCATED, entity X was not allocated in memory
                      (was optimized away)

%DEBUG-W-NOVALATPC, entity X does not have a value at the
                    current PC
```

次の Pascal の例は，この状況が発生する様子を示しています。

```
PROGRAM DOC(OUTPUT);
   VAR
      X,Y: INTEGER;
   BEGIN
      X := 5;
      Y := 2;
      WRITELN(X*Y);
   END.
```

このプログラムを/NOOPTIMIZE，またはそれに等しい修飾子を付けてコンパイルすると，デバッグのときに次のような正常な動作が得られます。

```
$ PASCAL/DEBUG/NOOPTIMIZE DOC
$ LINK/DEBUG DOC
$ DEBUG/KEEP
   .
   .
   .
DBG> RUN DOC
   .
   .
   .
DBG> STEP
stepped to DOC\%LINE 5
     5:         X := 5;
DBG> STEP
stepped to DOC\%LINE 6
     6:         Y := 2;
DBG> STEP
stepped to DOC\%LINE 7
     7:         WRITELN(X*Y);
DBG> EXAMINE X,Y
DOC\X:  5
DOC\Y:  2
DBG>
```

このプログラムを/OPTIMIZE，またはそれに等しい修飾子を付けてコンパイルすると， X と Y の値は初期値の割り当て以降は変更されないため，コンパイラは X*Y を計算して値 (10) を保存し， X と Y の記憶域を割り当てません。そのため，デバッグの開始後， STEP コマンドは第 5 行ではなく直接第 7 行に移ります。さらに， X や Y を検査することはできません。

```
$ PASCAL/DEBUG/OPTIMIZE DOC
$ LINK/DEBUG DOC
$ DEBUG/KEEP
   .
   .
   .
DBG> RUN DOC
   .
   .
   .
DBG> EXAMINE X,Y

%DEBUG-W-UNALLOCATED, entity X was not allocated in memory
                      (was optimized away)

DBG> STEP
stepped to DOC\%LINE 7
     7:         WRITELN(X*Y);
DBG>
```

これに対して，次の行は WRITELN 文の最適化されていないコードを表示します。

```
DBG> STEP
stepped to DOC\%LINE 7
     7:         WRITELN(X*Y);
DBG> EXAMINE/OPERAND .%PC

DOC\%LINE 7:    MOVL    S^#10,B^-4(FP)
     B^-4(FP)   2146279292 contains 62914576
DBG>
```

14.1.2 コーディング順序の変更

最適化の手法の中には，ソース・コードで指定されたシーケンスとは異なるシーケンスで操作を行うという方法をとるものがあります。場合によってはコードがすべて削除されてしまうこともあります。

その結果，デバッガが表示するソース・コードと，実際に実行されているコードが完全には対応しなくなります。

例を使用して説明するために， Fortran プログラムのソース・コードのセグメントを示します。これはコンパイラ・リストや画面モードのソース表示に表示されるものです。このコード・セグメントは配列 A の最初の 10 個の要素を 1/X の値に設定します。

```
Line          Source Code
----          -----------
  5            DO 100 I=1,10
  6            A(I) = 1/X
  7       100 CONTINUE
```

最適化は次の手順で行われます。コンパイラがソース・プログラムを処理するとき，X の逆数はソース・コードで指定されている 10 回ではなく， 1 回だけ計算すればよいと判断されます。 X の値は DO ループの中では変化しないからです。コンパイラはこのように，次に示すコード・セグメントと等しい最適化されたコードを作成します。

```
Line          Optimized Code Equivalent
----          -------------------------
  5            TEMP = 1/X
               DO 100 I=1,10
  6            A(I) = TEMP
  7       100 CONTINUE
```

コンパイラの機能によって，移動したコードがループの第 1 行と関連したり，元の行番号を維持したり (Alpha システムで共通) することがあります。

相違が発生しても，その相違は表示されるソース行を見るだけでは明らかではありません。さらに， X がゼロのために 1/X の計算が失敗したとき，ソース表示を入念に調べることによって，除算が全く含まれていないソース行でゼロによる除算が行われたことが明らかになります。

このようなソース・コードと実行可能なコードの明らかな不一致は，最適化されたプログラムをデバッグするときにしばしば見られます。前の例のようなループから抜ける場合のコード動作が原因になるだけでなく，他の多くの最適化の手法も原因になります。

14.1.3 セマンティク・ステップ実行 (Alpha のみ)

セマンティク・ステップ実行 (Alpha システムのみで使用可能) を行うことにより，最適化されたコードをステップ実行するとき，混乱を少なくすることができます。セマンティク・ステップ実行モードは，従来の行単位ステップ・モード，命令単位ステップ・モードを補足するものです。セマンティク・ステップ実行に使用するコマンドとして， SET STEP SEMANTIC_EVENT と STEP/SEMANTIC_EVENT の 2 種類のコマンドがあります。

◆セマンティク・イベント

最適化されたコードをステップ実行するときの問題として，明らかなソース・プログラムの位置が前後に「動き回り」，同じ行が何度も現れるということがあります。実際， STEP LINE モードでコードを前方向に実行している場合でも， STEP コマンドのたびに同じ命令が繰り返されることもあります。

この問題は，**セマンティク・イベント**の注釈命令によって対応できます。セマンティク・イベントは，次の 2 つの理由で重要です。

- プログラムの効果が実際に発生する箇所を示す。
- これらの効果が，プログラム内のイベントの順序と非常に近い順序で発生する。

セマンティク・イベントは，以下のいずれかになります。

- データ・イベント — ユーザ変数への割り当て
- 制御イベント — 呼び出しを除く，制御の条件転送，非条件転送による制御フローの決定
- 呼び出しイベント — (ステップ実行されないルーチンの) 呼び出しまたは呼び出しからの復帰

ただしセマンティク・イベントは，必ずしもイベント割り当て，制御の転送，呼び出しに限られるものではありません。例外として次のようなものがあります。

- 複合値または X_floating 値に 2 つの命令を割り当てる必要があるとき，最初の命令だけがセマンティク・イベントとして扱われる。

- case 構成体や select 構成体を持つ分岐の判断ツリーのように，1 つの高レベル構成体にそれを構成する分岐が複数ある場合，最初の分岐だけがセマンティク・イベントとして扱われる。
- ある種の文字列や記憶域のコピー操作を処理する OTS$MOVE を呼び出す場合のように，コンパイラ固有のヘルパ・ルーチンになっているルーチンを呼び出すとき，この呼び出しはセマンティク・イベントとみなされない。つまり，呼び出しの場合でも制御は停止しない。

  このようなルーチンにステップ移動するときは，次のいずれかを行う。

  – ルーチンのエントリ・ポイントにブレークポイントをセットする。
  – 該当する呼び出しになるまで一連の STEP/INSTRUCTION コマンドを使用し，そのあと STEP/INSTRUCTION/INTO コマンドを使用して，呼び出される側のルーチンに入る。
- ある行の中に複数のセマンティク・イベントがあって，同じ行番号が使用されているとき，最初のイベントのみが使用される。

◆ SET STEP SEMANTIC_EVENT コマンド

SET STEP SEMANTIC_EVENT コマンドは，省略時のステップ実行モードをセマンティクとして確立します。

◆ STEP/SEMANTIC_EVENT コマンド

STEP/SEMANTIC_EVENT コマンド (セマンティク・モードになっているときは単に STEP でも同等) は，次のセマンティク・イベントが割り当て，制御の転送，呼び出しのいずれであっても，そのイベントにブレークポイントをセットします。そのイベントに移るまで，実行が進みます。さまざまな行/文があれば，そのうちのいくつかがプロセスを阻害しない方法で実行されます。セマンティク・イベントに移る (つまりそのイベントに対応する命令に移ったが，まだ実行されていない状態) と，実行が停止します (STEP/LINE を使用しているときに次の行に移るのと同様)。

◆セマンティク・ステップ実行の例

次に示す C プログラム doct2 の内容は，最適化の場合の注意事項を示しています。

```
#include <stdio.h>
#include <stdlib.h>

int main(unsigned argc, char **argv) {
    int w, x, y, z=0;

    x = atoi(argv[1]);
    printf("%d\n", x);

    x = 5;
    y = x;

    if (y > 2) {                /* always true */
        printf("y > 2");
        }
    else {
        printf("y <= 2");
        }
```

```
if (z) {                    /* always false */
    printf("z");
    }
else {
    printf("not z");
    }

printf("\n");
}
```

次の 2 つの例は，最適化した doct2 プログラムによって，行単位でステップ実行した場合の例と，セマンティク・イベントによりステップ実行した場合の例で，上記と対照をなしています。

- 行単位のステップ実行

```
$ doct2:=$sys$disk:[]doct2
$ doct2 6

         Debugger Banner and Version Number

Language:: Module: Doct2: GO to reach DBG> go
break at routine DOCT2\main
   654:     x = atoi(argv[1]);
DBG> step
stepped to DOCT2\main\%LINE 651
   651: int main(unsigned argc, char **argv) {
DBG> step
stepped to DOCT2\main\%LINE 654
   654:     x = atoi(argv[1]);
DBG> step
stepped to DOCT2\main\%LINE 651
   651: int main(unsigned argc, char **argv) {
DBG> step
stepped to DOCT2\main\%LINE 654
   654:     x = atoi(argv[1]);
DBG> step
stepped to DOCT2\main\%LINE 655
   655:     printf("%d\n", x);
DBG> step
stepped to DOCT2\main\%LINE 654
   654:     x = atoi(argv[1]);
DBG> step
stepped to DOCT2\main\%LINE 655
   655:     printf("%d\n", x);
DBG> step
6
stepped to DOCT2\main\%LINE 661
   661:         printf("y > 2");
DBG> step
y > 2
stepped to DOCT2\main\%LINE 671
   671:         printf("not z");
DBG> step
not z
```

```
stepped to DOCT2\main\%LINE 674
   674:     printf("\n");
DBG> step
stepped to DOCT2\main\%LINE 675
   675:     }
DBG> step
'Normal successful completion'
DBG>
```

- セマンティク・イベントによるステップ実行

```
$ doct2:=$sys$disk:[]doct2
$ doct2 6

         Debugger Banner and Version Number

Language:: Module: Doct2: GO to reach DBG> set step semantic_event
DBG> go
break at routine DOCT2\main
   654:     x = atoi(argv[1]);
DBG> step
stepped to DOCT2\main\%LINE 654+8
   654:     x = atoi(argv[1]);
DBG> step
stepped to DOCT2\main\%LINE 655+12
   655:     printf("%d\n", x);
DBG> step
6
stepped to DOCT2\main\%LINE 661+16
   661:         printf("y > 2");
DBG> step
y > 2
stepped to DOCT2\main\%LINE 671+16
   671:         printf("not z");
DBG> step
not z
stepped to DOCT2\main\%LINE 674+16
   674:     printf("\n");
DBG> step
stepped to DOCT2\main\%LINE 675+24
   675:     }
DBG> step
stepped to DOCT2\__main+104
DBG> step
'Normal successful completion'
DBG>
```

セマンティク・ステップ実行の動作は，行単位のステップ実行と比べてはるかに滑らかで，直線的な方法になります。またセマンティク・ステップ実行では，問題のある重要な箇所で停止します。一般的に，最適化したコードを行単位でステップ実行する場合に特徴的な，コードが前後に「動き回」ることによる混乱は，セマンティク・ステップ実行の場合，激減するかまたは完全になくなります。ソース・プログラムの順序変更により，実行特性が向上することもありますが，通常フローは上から下に向かって移動します。

ステップ実行の密度は，行単位のステップ実行とセマンティク・ステップ実行とで異なります。行単位の方が大きくなることもあり，セマンティクの方が大きくなることもあります。たとえば，セマンティクの性質によりセマンティク・イベントを構成する文は，その文が完全に最適化されている場合，セマンティク・ステップ実行で出てくることはありません。このように，セマンティク・リージョンは複数の行にまたがっており，最適化された行はスキップされるということになります。

14.1.4 レジスタの使用

コンパイラは， 1 つの式の値は 2 つの出現の間では変化しないものと判断し，その値をレジスタに保存することがあります。このような場合，コンパイラは次の出現では値の再計算は行わず，レジスタに保存されている値が有効であるとみなします。

プログラムをデバッグしている間に，式中の変数の値を変更するために DEPOSIT コマンドを使用しても，レジスタに保存されている対応する値は変更されません。したがって，処理を続行する場合，式中の変更された値ではなくレジスタの値が使用され，予期しない結果になることがあります。

さらに，非静的変数 (第 3.4.3 項を参照) の値がレジスタに保存されている場合，メモリ内にあるその値は通常，無効になります。したがって，このような状況下では，変数に対して EXAMINE コマンドを入力すると誤った値が表示されることがあります。

14.1.5 存在期間分割変数

最適化してコンパイルするとき，コンパイラが変数を，別個に割り当てることのできるいくつかの部分変数に「分割」して，その変数で存在期間分割による分析を実行することがあります。その結果，元の変数が，別々の時間に別々の場所 (場合によってはレジスタ，場合によってはメモリ，場合によってはどこにもない) に置かれているということも可能になります。異なる部分変数を使用すれば，それぞれの変数を同時にアクティブにすることもできます。

Alpha プロセッサでは， EXAMINE コマンドを使用すると，プログラムのどの場所で変数が定義されているかが表示されます。変数が不適切な値のとき，この位置の情報により，変数の値がどこで割り当てられているか判断することができます。また/DEFINITION S 修飾子により，この位置の数を省略時の 5 から別の値に変更できるようになっています。

存在期間分割による分析は，スカラ変数およびパラメータのみに適用されます。配列，レコード，構造体，その他の集合体には適用されません。

◆存在期間分割処理の例

次の例は，存在期間分割処理の例を示しています。最初の例は，小さなCプログラムですが，ここでは，左の段の数字が行番号を表しています。

```
385 doct8 () {
386
387     int i, j, k;
388
389     i = 1;
390     j = 2;
391     k = 3;
392
393     if (foo(i)) {
394         j = 17;
395         }
396     else {
397         k = 18;
398         }
399
400     printf("%d, %d, %d\n", i, j, k);
401
402     }
```

最適化されているプログラムで，デバッグのために，コンパイル，リンク，実行を行うとき，次のようなダイアログが表示されます。

```
$ run doct8

   .
   .
   .
     DBG> step/into
     stepped to DOCT8\doct8\%LINE 391
        391:     k = 3;
     DBG> examine i
     %W, entity 'i' was not allocated in memory (was optimized away)
     DBG> examine j
     %W, entity 'j' does not have a value at the current PC
     DBG> examine k
     %W, entity 'k' does not have a value at the current PC
```

変数iのメッセージが，変数j，kのメッセージと異なることに注意してください。変数iは，メモリ(レジスタ，コア，その他)に全く割り当てられていないため，その値を再度テストする意味はありません。比較のために見てみると，jとkには，この時点で「現在のPC」に値がありません。これ以降のプログラムの中で出てきます。

もう1行ステップ実行すると，次のようになります。

```
     DBG> step
     stepped to DOCT8\doct8\%LINE 385
        385: doct8 () {
```

これを見ると，1 ステップ戻っているように見えます。最適化 (スケジューリング) したコードで共通の現象です。この問題については，第 14.1.2 項の「セマンティク・ステップ実行モード」の項で説明しています。さらにステップ実行を続けると，次のようになります。

```
DBG> step 5
stepped to DOCT8\doct8\%LINE 391
   391:     k = 3;
DBG> examine k
%W, entity 'k' does not have a value at the current PC
DBG> step
stepped to DOCT8\doct8\%LINE 393
   393:     if (foo(i)) {
DBG> examine j
%W, entity 'j' does not have a value at the current PC
DBG> examine k
DOCT8\doct8\k:  3
    value defined at DOCT8\doct8\%LINE 391
```

ここではjがまだ定義されていませんが，kには 3 という値があります。この値は 391 行目で割り当てられているものです。

ソースでは，kの前にjに値が割り当てられている (390 行目) ため，これはすでに表示されていなければなりません。ここでも最適化 (スケジューリング) したコードで共通の現象が発生しています。

```
DBG> step
stepped to DOCT8\doct8\%LINE 390
   390:     j = 2;
```

ここでjの値が表示されます。

```
DBG> examine j
%W, entity 'j' does not have a value at the current PC
DBG> step
stepped to DOCT8\doct8\%LINE 393
   393:     if (foo(i)) {
DBG> examine j
DOCT8\doct8\j:  2
    value defined at DOCT8\doct8\%LINE 390
```

400 行目の print 文までスキップし，jの値をテストします。

```
DBG> set break %line 400
DBG> g
break at DOCT8\doct8\%LINE 400
   400:     printf("%d, %d, %d\n", i, j, k);
DBG> examine j
DOCT8\doct8\j:  2
    value defined at DOCT8\doct8\%LINE 390
    value defined at DOCT8\doct8\%LINE 394
```

jの値を定義した位置が複数あります。 IF 節で選択されるパスにより，どちらかの位置が適用されます。この機能を使用すれば，変数が明らかに不適切な値の場合，その場所を調べることができ，同時に値がどこから来ているのかについても調べることができます。

次の例のように SHOW SYMBOL/ADDRESS コマンドを使用すると，あるシンボルの存在期間分割情報を表示することができます。

```
DBG> show symbol/address j
     data DOCT8\doct8\j
       between PC 131128 and 131140
         PC definition locations are at: 131124
         address: %R3
       between PC 131144 and 131148
         PC definition locations are at: 131140
         address: %R3
       between PC 131152 and 131156
         PC definition locations are at: 131124
         address: %R3
       between PC 131160 and 131208
         PC definition locations are at: 131124, 131140
         address: %R3
```

変数jには， 4 つの存在期間セグメントがあります。 PC アドレスは，イメージのリンク結果を示し，同時にソース・プログラムの行番号との対応が，コメントにより示されます。

- 最初のセグメントは， 2 をjに割り当てる箇所で開始しており， 17 をjに割り当てる直前までテストの間継続している。
- 2 番目のセグメントは， 17 をjに割り当てる箇所で開始しており， IF 文の ELSE 部分まで継続している。
- 3 番目のセグメントは， ELSE 節に対応している。この範囲の PC では，jへの割り当てはない。ここで適用されているjの定義は，最初のセグメントのものになる。
- 4 番目のセグメントは， IF 文の後の結合の箇所で開始しており，プログラムの最後まで継続している。jの定義は， 390 行目または 394 行目から来ている。どちらの行になるかは， IF 文のときどちらのパスが選択されたかによって異なる。

Alpha システムの場合，デバッガは，どの割り当てとどの定義が変数の表示値を規定するかについて，トラックして報告します。この補足情報を使用することにより，コードのモーションなどの最適化の効果のうちいくつかを処理できるようになります。変数値がプログラムの予期しない場所から来ているような場合も，その効果を処理できるようになります。

◆ EXAMINE/DEFINITIONS コマンド (Alpha のみ)
存在期間分割変数の場合， EXAMINE コマンドは，アクティブな存在期間の値を表示するだけでなく，存在期間の定義位置も表示します。定義位置とは，存在期間が初期値を受け取る場所を表します。定義位置が 1 つの場合，それが唯一の位置になります。

存在期間の初期値が複数の位置から来ている場合，定義位置も複数になります。前の例では，プログラムがprintf,で停止したときにjをテストすると次のようになります。

```
DBG> examine j
DOCT8\doct8\j:  2
    value defined at DOCT8\doct8\%LINE 390
    value defined at DOCT8\doct8\%LINE 394
```

この場合， 393 行目の式が TURE かどうかにより，値が 390 行目と 394 行目のどちらから来るか決まるため，jの存在期間には， 2 つの定義位置があることになります。

省略時の場合，変数の内容がテストされるときに最大で 5 つの定義位置が表示されます。定義位置の数を指定するときは，次の例のように/DEFINITIONS= *n*修飾子を使用します。

```
DBG> EXAMINE/DEFINITIONS=74 FOO
```

省略形は/DEFI になります。

省略時の定義数を 5 以外の数にしたい場合，次のようなコマンド定義を使用します。

```
DBG> DEFINE/COMMAND E = "EXAMINE/DEFINITIONS=100"
```

/DEFINITIONS 修飾子に 100 が設定されていて，テストしている存在期間分割変数に 120 の定義位置がある場合，デバッガは，指定に従って 100 個表示しますが，同時に次のようなレポートも表示します。

```
there are 20 more definition points
```

14.2 画面用プログラムのデバッグ

デバッグ・セッションの間，デバッガは端末の画面を入出力 (I/O) のために使用します。 1 台の端末を使用して，画面の大部分または全体を使用するような画面用プログラムのデバッグを行う場合，デバッガの入出力はプログラムの入出力に対して上書きしたり上書きされたりします。

1 台の端末をプログラムの入出力とデバッガの入出力のために使用すると，画面モードでデバッグし画面用プログラムが実行時ライブラリ (RTL) 画面管理 (SMG$) ルーチンのどれかを呼び出している場合はさらに複雑になります。これは，デバッガの画面モードも SMG ルーチンを呼び出すからです。このような場合，デバッガとユ

ーザ・プログラムは同じ SMG ペーストボードを共用し，一層の混乱を引き起こします。

画面用プログラムをデバッグする際のこのような問題を回避するために，デバッガの入出力をプログラムの入出力から分離するために次の方法のいずれかを使用します。

- VWS を使用中のワークステーションの場合，デバッグ・セッションを開始し，デバッガ・コマンドの SET MODE SEPARATE を入力する。このコマンドはデバッガの入出力用として分離されたターミナル・エミュレータ・ウィンドウを作成する。プログラムの入出力はデバッガの起動ウィンドウにそのまま表示される。
- DECwindows Motif を使用中のワークステーションの場合は次のようになる。
  - 別のワークステーション (同じように DECwindows Motif を使用中のもの) 上にデバッガの DECwindows Motif インタフェースを表示する方法については，第 9.8.3.1 項を参照。
  - DECwindows Motif インタフェースではなくデバッガのコマンド・インタフェースを使用する方法については，第 9.8.3.3 項を参照。デバッガの入出力用として分離した DECterm ウィンドウを作成する方法が説明されている。この方法は， VWS を使用中のワークステーション上で SET MODE SEPARATE コマンドを使用した場合と同じ効果を持つ。
- ワークステーションがない場合は， 2 つの端末を使用し，そのうちの 1 つをプログラムの入出力用に，もう 1 つをデバッガの入出力用に使用する。以降はこの方法について説明する。

TTD1: が現在の端末であり，そこからデバッグを開始するものとします。デバッガの入出力は端末 TTD2: に表示し， TTD1: はプログラムの入出力専用にするものとします。

次の手順に従ってください。

1. TTD1: から TTD2 を占有できるよう， TTD2: に必要な保護を設定する (詳細は第 14.2.1 項を参照)。

   以降の操作はすべて TTD1: で行う。

2. TTD2: を占有する。この結果，次のようにユーザの処理は TTD1: で行われ，TDD2: へは排他的アクセスが可能になる。

   ```
   $ ALLOCATE TTD2:
   ```

3. 次のようにデバッガの論理名 DBG$INPUT および DBG$OUTPUT を TTD2: に割り当てる。

   ```
   $ DEFINE DBG$INPUT TTD2:
   $ DEFINE DBG$OUTPUT TTD2:
   ```

DBG$INPUT および DBG$OUTPUT は，デバッガの入力装置および出力装置をそれぞれ指定する。省略時には，これらの論理名はそれぞれ SYS$INPUT および SYS$OUTPUT になる。 DBG$INPUT および DBG$OUTPUT を TDD2: に割り当てることによって，デバッガ・コマンドおよびデバッガの出力を TDD2: に表示できるようになる。

4. システムが端末のタイプを認識していることを確認する。次のコマンドを入力する。

```
$ SHOW DEVICE/FULL TTD2:
```

装置タイプが unknown のとき，システム管理者または LOG_IO 特権か PHY_IO 特権を持つユーザは次の例のような方法でシステムに対して端末タイプを認識させる必要がある。この例では，端末を VT200 としている。

```
$ SET TERMINAL/PERMANENT/DEVICE=VT200 TTD2:
```

5. デバッグするプログラムを実行する。

```
$ DEBUG/KEEP
   .
   .
   .
DBG> RUN prog-name
```

デバッガの入出力を TTD2: で観察することができる。

6. デバッグ・セッションを終了したら， TTD2: の占有を解除する。次のように行う。またはログアウトする。

```
$ DEALLOCATE TTD2:
```

14.2.1 端末を占有するための保護の設定

適切に保護されているシステムでは，ある端末から別の端末を占有することができないように，端末が保護されています。

必要な保護を設定するために，システム管理者または必要な特権を持つユーザは次の例のような手順に従う必要があります。

この例では， TTD1: が現在の端末 (デバッガを起動する端末)， TTD2: がデバッガの入出力を表示するために占有する端末です。

1. TTD1: と TTD2 の両方がシステムに物理的に接続されている場合は，ステップ 4 に移る。

TTD1: と TTD2: が LAT (ローカル・エリア・トランスポート) を介してシステムに接続されている場合は，ステップ 2 に進む。

2. TTD2: にログインする。

3. 次のコマンドを入力する。 LOG_IO 特権または PHY_IO 特権が必要である。

```
$ SET PROCESS/PRIV=LOG_IO
$ SET TERMINAL/NOHANG/PERMANENT
$ LOGOUT/NOHANG
```

4. 次のいずれか 1 つのコマンドを入力する。 OPER 特権が必要である。

```
$ SET ACL/OBJECT_TYPE=DEVICE/ACL=(IDENT=[PROJ,JONES],ACCESS=READ+WRITE) TTD2:   1
$ SET PROTECTION=WORLD:RW/DEVICE TTD2:   2
```

1 SET ACL コマンド行はアクセス制御リスト (ACL) を使用するので，このコマンドを使用するのがよい。この例では，アクセスはユーザ識別コード (UIC) [PROJ,JONES]に制限されている。

2 SET PROTECTION コマンド行は，ワード読み込み/書き込みアクセスを許可する。したがって，どのユーザも TTD2: を占有し，そこで入出力を実行することが可能になる。

14.3 複数言語プログラムのデバッグ

1 つの同じデバッグ・セッションの中で，ソース・コードが異なる言語で記述されたモジュールをデバッグすることができます。言語固有の動作について，混乱を避けるために注意すべきことを中心に説明します。

どの言語の場合でも，デバッグの際には次を参照してください。

- デバッガのオンライン・ヘルプ (HELP Languageと入力する)。
- その言語とともに提供されているドキュメント。

14.3.1 現在のデバッガ言語の制御

プログラムをデバッガの制御下に置くとき，デバッガは**現在の言語**をメイン・プログラムが含まれているモジュール (通常は，イメージ転送アドレスを含んでいるルーチン) を記述している言語に設定します。現在の言語は，その時点で識別されます。次に例を示します。

```
$ DEBUG/KEEP

          Debugger Banner and Version Number

DBG> RUN prog-name
Language: PASCAL, Module: FORMS
DBG>
```

現在の言語の設定は，デバッガ・コマンドで指定した名前，演算子，式をデバッガがどう解析し，解釈するかを決定します。変数の型の設定，配列やレコードの構文，整数データの省略時の基数，大文字/小文字の区別などの解釈も含まれます。言語の設定によって，ユーザ・プログラムに関連するデータの表示方法も決まります。

多くのプログラムでは，メイン・プログラムの言語とは別の言語で記述されたモジュールが含まれています。混乱をできるだけ少なくするために，省略時の設定ではデバッガの言語は，別の言語で記述されたモジュールの中で実行が一時停止しても，デバッグ・セッションを通してメイン・プログラムの言語に設定されたまま変わりません。

このようなモジュールでのシンボリック・デバッグの利点を最大に活用するためには， SET LANGUAGE コマンドを使用してデバッグ・コンテキストを別の言語のデバッグ・コンテキストに設定します。たとえば次のコマンドは，デバッガがシンボルや式などを COBOL 言語の規則に従って解釈するように設定します。

```
DBG> SET LANGUAGE COBOL
```

Alpha プロセッサでは， SET LANGUAGE コマンドは次のキーワードを受け付けます。

Ada	BASIC	BLISS	C
C++	COBOL	Fortran	MACRO-32[1]
MACRO–64	Pascal	PL/I	

[1]MACRO-32 は AMACRO コンパイラでコンパイルしなければなりません。

Integrity プロセッサでは， SET LANGUAGE コマンドは次のキーワードを受け付けます。

Assembler (IAS)	BASIC	BLISS	C
C++	COBOL	Fortran	MACRO-32[1]
IMACRO	PASCAL		

[1]MACRO-32 は AMACRO コンパイラでコンパイルしなければならない点に注意してください。

さらに，サポートされていない言語で記述されたプログラムをデバッグする場合， SET LANGUAGE UNKNOWN コマンドを指定することができます。サポートされていない言語でもデバッガを最大限活用できるようにするために， SET LANGUAGE UNKNOWN コマンドはデバッガがデータ形式と演算子の組み合わせを広く受け付けるようにします。これらの中には，サポートされている言語の少数にだけ固有のものも含まれます。言語が UNKNOWN に設定された場合に認識される演算子や構造については，デバッガのオンライン・ヘルプに示します (HELP Languageと入力します)。

14.3.2 言語に固有の相違点

ここでは，各言語でデバッグを行う場合に注意すべき相違点を説明します。 SET LANGUAGE コマンドの影響を受ける相違点およびその他の相違点，たとえば，言語固有の初期化コードや定義済みのブレークポイントなどが挙げられています。

ここに挙げられているものはすべてではありません。詳細については，デバッガのオンライン・ヘルプ (HELP Languageと入力する) および使用している言語のドキュメントを参照してください。

14.3.2.1 省略時の基数

整数データを入力したり表示したりするための省略時の基数は，ほとんどの言語では 10 進数です。

Alpha プロセッサでは， BLISS, MACRO–32 および MACRO–64 の場合は例外であり，これらは 16 進数を省略時の基数とします。

新しい省略時の基数を設定するには， SET RADIX コマンドを使用します。

14.3.2.2 言語式の評価

いくつかのデバッガ・コマンドや構造は言語式を評価します。

- EVALUATE， DEPOSIT， IF， FOR， REPEAT， WHILE の各コマンド
- SET BREAK， SET TRACE， SET WATCH の各コマンドとともに使用される WHEN 句。

これらのコマンドを処理するとき，第 4.1.6 項で説明されているように，デバッガは現在の言語の構文および現在の基数に基づいて言語式を評価します。デバッガは，(コマンドを入力するときではなく) 実行のたびに， WHEN 節や DO 節の式の構文をチェックし，その後これらを評価します。

演算子は言語によって大きく異なりますので注意してください。たとえば，次の 2 つのコマンドはそれぞれ Pascal および Fortran で記述された同じ意味の式を評価しています。

```
DBG> SET WATCH X WHEN (Y < 5)       ! Pascal
DBG> SET WATCH X WHEN (Y .LT. 5)    ! FORTRAN
```

言語が PASCAL に設定されていて，最初の Pascal のコマンドを入力したとします。ここで Fortran のルーチン内の命令をステップ実行して言語を Fortran に設定し，処理を再開します。言語が Fortran に設定されている間，デバッガは式 (Y < 5) を評価することができません。その結果，無条件のウォッチポイントを設定します。ウォッチポイントが検出されると，<演算子に対する構文エラーが返されます。

このような矛盾は，デバッガ・コマンド・プロシージャやデバッガ初期化ファイル内で言語式を評価するコマンドを実行する場合にも発生します。

言語が BLISS に設定された場合，デバッガは変数名または他のアドレス式を含む言語式を別の言語が設定された場合とは異なるように処理します。詳細については第 4.1.6 項を参照してください。

14.3.2.3 配列およびレコード

配列要素やレコード構成要素 (適用できる場合) を表す文法は，言語によって異なります。

たとえば，ある言語では配列要素を区切るために大括弧 ([ ]) を使用し，別の言語では括弧 (( )) を使用します。

ある言語ではゼロを基底とする配列を持ち，また別の言語では，次の例のように 1 を基底とする配列を持ちます。

```
DBG> EXAMINE INTEGER_ARRAY
PROG2\INTEGER_ARRAY
   (1,1):       27
   (1,2):       31
   (1,3):       12
   (2,1):       15
   (2,2):       22
   (2,3):       18
DBG>
```

ある言語 (Pascal や Ada など) では，特定の配列宣言によって配列が何を基底にするかが決まります。

14.3.2.4 大文字/小文字の区別

C 言語では，名前や言語式の大文字/小文字が区別されます。名前や言語式を指定する場合，ソース・コードに記述されているとおりに指定しなければなりません。たとえば，言語が C に設定されているとき，次の 2 つのコマンドは等しくありません。

```
DBG> SET BREAK SCREEN_IO\%LINE 10
DBG> SET BREAK screen_io\%LINE 10
```

14.3.2.5 初期化コード

多くのプログラムでは，プログラムがデバッガの制御下に置かれたとき，NOTATMAIN メッセージが表示されます。次に例を示します。

```
$ DEBUG/KEEP

          Debugger Banner and Version Number

DBG> RUN prog-name
Language: ADA, Module: MONITOR
Type GO to reach main program
DBG>
```

NOTATMAIN メッセージは，メイン・プログラムの先頭の前でプログラムが一時停止していることを知らせます。これによって，デバッガの制御下でいくつかの初期化コードを実行し，チェックすることが可能になります。

初期化コードはコンパイラによって作成され， LIB$INITIALIZE という名前の特別な PSECT に置かれます。たとえば， Ada パッケージの場合，初期化コードはパッケージ本体 (変数を初期化するための文を含む) に属します。 Fortran プログラムの場合

は，初期化コードは/CHECK=UNDERFLOW 修飾子または/CHECK=ALL 修飾子を指定したときに必要とされるハンドラを宣言します。

NOTATMAIN メッセージは，初期化コードをデバッグしたくない場合， GO コマンドを入力すると直ちにメイン・プログラムの先頭から実行できることを知らせます。このとき，ユーザは他のプログラムのデバッグを起動するときと同じ場面にいます。GO コマンドを再び入力すると，プログラムの実行が開始されます。

14.3.2.6 定義済みのブレークポイント

タスキング・プログラムの場合，プログラムがデバッガの制御下に置かれると，タスキング例外イベントに関連づけられた 2 つのブレークポイントが自動的に設定されます。これらのブレークポイントは， SET LANGUAGE コマンドの影響を受けません。これらのブレークポイントは，適切な実行ライブラリが存在すると，デバッガの初期化の間に自動的に設定されます。

これらの定義済みのブレークポイントを示すには， SHOW BREAK コマンドを入力します。次に例を示します。

```
DBG> SHOW BREAK
Predefined breakpoint on ADA event "EXCEPTION_TERMINATED" for any value
Predefined breakpoint on ADA event "DEPENDENTS_EXCEPTION" for any value
DBG>
```

14.4 スタックの破損からの回復

デバッガは，起動時に一定量のメモリを割り当て，ユーザのプログラムとスタックを共有します。ユーザ・プロセスの例外により，リソースの浪費やスタックの破損が発生した場合，デバッガは制御を失うことがあり，その場合デバッグ・セッションも異常終了します。

スタック破損のメッセージや，重大なエラーについての警告が出たら，このような事態になることも考えておかなければなりません。どちらの場合も，デバッグ・セッションの完全性が保証されなくなります。

次のいずれかの手段を試してみます。

- リソースの消費量を減らすか，スタック領域の使用量を小さくするため，ソース・コードを一時的または永続的に変更する。
- 割り当て量を増やす。
- プログラムのリンク時に大きなスタック・サイズを指定する。

14.5 例外ハンドラおよび条件ハンドラのデバッグ

条件ハンドラは，例外が発生した場合にオペレーティング・システムが実行するプロシージャです。

例外には，ハードウェア条件 (算術演算でのオーバフローやメモリ・アクセス違反など)，またはシグナル通知されたソフトウェア的な例外 (ファイルが見つからないためにシグナル通知された例外など) が含まれます。

オペレーティング・システムやデバッガ，またはユーザ・プログラムによって設定された種々の条件ハンドラ，たとえば，1 次ハンドラ，呼び出しフレーム・ハンドラ (アプリケーションで宣言されたもの) などが，どのように，またどんな順序で起動されるかは，オペレーティング・システムの規則によって指定されます。デバッガを使用する場合の条件処理については，第 14.5.3 項を参照してください。条件処理についての一般的な説明については，『OpenVMS Run-Time Library Routines Volume』を参照してください。

例外ハンドラおよび条件ハンドラのデバッグ・ツールには次のものがあります。

- SET BREAK/EXCEPTION コマンドおよび SET TRACE/EXCEPTION コマンド。これらのコマンドは，ユーザ・プログラムによって発生した例外をそれぞれブレークポイントまたはトレースポイントとしてデバッガが扱うようにする。第 14.5.1 項および第 14.5.2 項を参照。
- いくつかの組み込みシンボル (たとえば，%EXC_NAME)。これらは例外ブレークポイントおよび例外トレースポイントを修飾する。第 14.5.4 項を参照。
- SET BREAK/EVENT コマンドおよび SET TRACE/EVENT コマンド。これらのコマンドは，Ada，SCAN，およびマルチスレッド・プログラムなどに固有の例外イベントでのブレークまたはトレースを行う。詳しい説明は，対応するドキュメントを参照。

14.5.1 例外へのブレークポイントまたはトレースポイントの設定

SET BREAK/EXCEPTION または SET TRACE/EXCEPTION コマンドを入力すると，デバッガはユーザ・プログラムによって作成された例外をブレークポイントまたはトレースポイントとして扱うようになります。SET BREAK/EXCEPTION コマンドの結果，ユーザのプログラムが例外を作成した場合は，デバッガは実行を一時停止し，例外が生じたことと実行が一時停止した行を報告し，コマンド入力を要求するプロンプトを表示します。次に例を示します。

```
DBG> SET BREAK/EXCEPTION
DBG> GO
   .
   .
   .
%SYSTEM-F-INTDIV, arithmetic trap, integer divide by zero at PC=0000066C, PSL=03C00022
break on exception preceding TEST\%LINE 13
     6:         X := 3/Y;
DBG>
```

例外ブレークポイント (例外トレースポイント) は，例外を処理するための条件ハンドラをユーザ・プログラムが持っている場合でも検出されます。 SET BREAK /EXCEPTION コマンドは，ハンドラが実行可能になる前およびその結果，例外が破棄される前にブレークポイントを発生させます。例外ブレークポイントがなければハンドラは実行され，例外を破棄するハンドラがない場合にだけ，デバッガが制御を得ることができます。第 14.5.2 項および第 14.5.3 項を参照してください。

次のコマンド行は，例外が発生した場所を示すために使用すると便利です。このコマンドを使用すると，デバッガは一連のアクティブな呼び出しと例外ブレークポイントでの PC 値を自動的に表示します。

```
DBG> SET BREAK/EXCEPTION DO (SET MODULE/CALLS; SHOW CALLS)
```

画面モードの DO 表示を作成して，デバッガが実行に割り込みをかけた場合に SHOW CALLS コマンドを実行することもできます。次に例を示します。

```
DBG> DISPLAY CALLS DO (SET MODULE/CALLS; SHOW CALLS)
```

SET TRACE/EXCEPTION コマンドで設定される例外トレースポイントは，例外ブレークポイントにアドレス式の指定を持たない GO コマンドが続くものと似ています。

例外ブレークポイントは例外トレースポイントを取り消します。また，その逆も同様です。

例外ブレークポイントまたは例外トレースポイントを取り消すには，それぞれ CANCEL BREAK/EXCEPTION コマンドまたは CANCEL TRACE/EXCEPTION コマンドを使用します。

14.5.2 例外ブレークポイントでの実行の再開

例外ブレークポイントが検出されると，アプリケーションで宣言された条件ハンドラが起動される前に実行が一時停止します。ブレークポイントから GO， STEP，または CALL の各コマンドで実行を再開する場合，動作は次のようになります。

- GO コマンドをアドレス式のパラメータなしで入力するか，または STEP コマンドを入力すると，デバッガは例外を再シグナル通知する。 GO コマンドは，アプリケーションで宣言されたハンドラがある場合，次にどのハンドラが例外を処理

するかをユーザが観察できるようにする。 STEP コマンドは，そのハンドラ内の命令をステップ実行する。次の例を参照。

- GO コマンドにアドレス式のパラメータを付けて入力すると，指定した記憶位置で処理が再開され，アプリケーションで宣言されたハンドラの実行が禁止される。
- 例外ブレークポイントでの一般的なデバッグ方法は， CALL コマンドでダンプ・ルーチンを呼び出す方法である (第 13 章を参照)。例外ブレークポイントで CALL コマンドを入力する場合，呼び出されたルーチン内で前に設定されたブレークポイント，トレースポイント，ウォッチポイントはどれもアクティブではないので，デバッガは例外コンテキストを失うことはない。ルーチンの実行後，デバッガは入力を要求する。ここで GO コマンドまたは STEP コマンドを入力すると，デバッガは例外を再シグナル通知する。

次の FORTRAN の例では，例外ブレークポイントで条件ハンドラの存在をどのようにして判断するか，また，ブレークポイントで入力された STEP コマンドがどのようにしてハンドラ内の命令をステップ実行するかを示しています。

例外ブレークポイントでは， SHOW CALL コマンドが SYS$QIOW ルーチン呼び出しの間に例外が発生したことを知らせます。

```
DBG> SET BREAK/EXCEPTION
DBG> GO
   .
   .
   .
%SYSTEM-F-SSFAIL, system service failure exception, status=0000013C, PC=7FFEDE06, PSL=03C00000
break on exception preceding SYS$QIOW+6
DBG> SHOW CALLS
module name  routine name       line       rel PC    abs PC
             SYS$QIOW                         00000006  7FFEDE06
*EXC$MAIN    EXC$MAIN             23      0000003B  0000063B
DBG>
```

VAX プロセッサでは，次の SHOW STACK コマンドは， SYS$QIOW ルーチンではハンドラが宣言されていないことを示しています。呼び出しスタックの 1 レベル下で， EXC$MAIN ルーチンが「SSHAND」という名前のハンドラを宣言しています。

```
DBG> SHOW STACK
stack frame 0 (2146296644)
    condition handler: 0
       SPA:            0
       S:              0
       mask:           ^M<r2,r3,r4,r5,r6,r7,r8,r9,r10,r11>
       PSW:            0020 (hexadecimal)
    saved AP:          2146296780
    saved FP:          2146296704
    saved PC:          EXC$MAIN\%LINE 25
```

```
   .
   .
   .
stack frame 1 (2146296704)
    condition handler: SSHAND
       SPA:           0
       S:             0
       mask:          ^M<r11>
       PSW:           0000 (hexadecimal)
    saved AP:         2146296780
    saved FP:         2146296760
    saved PC:         SHARE$DEBUG+2217
   .
   .
   .
```

この例外ブレークポイントで STEP コマンドを入力すると，条件ハンドラ「SSHAND」内の命令を直接ステップ実行することができます。

```
DBG> STEP
stepped to routine SSHAND
     2:        INTEGER*4 FUNCTION SSHAND (SIGARGS, MECHARGS)
DBG> SHOW CALLS
 module name  routine name    line     rel PC    abs PC
*SSHAND       SSHAND            2    00000002  00000642
----- 上記の条件ハンドラは例外0000045Cで呼び出された。
%SYSTEM-F-SSFAIL, system service failure exception, status=0000013C, PC=7FFEDE06, PSL=03C00000
----- 例外メッセージの終わり。
              SYS$QIOW                  00000006  7FFEDE06
*EXC$MAIN     EXC$MAIN         23       0000003B  0000063B
DBG>
```

デバッガは，可能な場合には条件ハンドラのアドレスを名前としてシンボル化します。ただし，ある言語では，アプリケーションで宣言された条件ハンドラが起動される前に，例外はまず実行時ライブラリ (RTL) ルーチンによって処理されるので注意が必要です。このような場合，最初の条件ハンドラのアドレスは RTL 共用可能イメージのアドレスからのオフセットにシンボル化されます。

14.5.3 条件ハンドラへのデバッガの影響

プログラムをデバッガとともに実行する場合，次の条件ハンドラのうち少なくとも 1 つが，プログラムの実行によって発生した例外を処理するために起動されます。複数の場合は次のリストの順番に起動されます。

1. 1 次ハンドラ
2. 2 次ハンドラ
3. 呼び出しフレーム・ハンドラ (アプリケーションで宣言される)。スタック・ハンドラとしても知られる。
4. 最終ハンドラ

5. ラスト・チャンス・ハンドラ
6. キャッチオール・ハンドラ

ハンドラは，次の 3 つの状態コードのうち 1 つを「Condition Handling Facility」へ戻します。

- SS$_RESIGNAL— オペレーティング・システムが次のハンドラを検索する。
- SS$_CONTINUE— 条件は修正されたとみなされ，実行が続けられる。
- SS$_UNWIND— 必要な場合には，呼び出しスタックはいくつかのフレームが展開され，信号が破棄される。

条件処理についてさらに詳しい説明は，『OpenVMS Programming Concepts Manual』を参照してください。

14.5.3.1 1 次ハンドラ

プログラムをデバッガとともに実行する場合，1 次ハンドラはデバッガです。したがって，デバッガが例外を処理する最初の機会を持ちます。例外がデバッガによって引き起こされたものであるかどうかは関係ありません。

SET BREAK/EXCEPTION コマンドまたは SET TRACE/EXCEPTION コマンドを入力した場合，デバッガはユーザ・プログラムによって引き起こされた例外でブレーク (トレース) します。ブレーク (トレース) 処理は，アプリケーションで宣言されたハンドラが起動される前に行われます。

SET BREAK/EXCEPTION コマンドまたは SET TRACE/EXCEPTION コマンドを入力していない場合，1 次ハンドラはユーザ・プログラムによって引き起こされた例外を再シグナル通知します。

14.5.3.2 2 次ハンドラ

2 次ハンドラは，特別な目的で使用され，本書で説明しているようなプログラムに対しては適用されません。

14.5.3.3 呼び出しフレーム・ハンドラ (アプリケーションで宣言されたもの)

ユーザ・プログラムの各ルーチンで，条件ハンドラを設定することができます。これらは呼び出しフレーム・ハンドラとして知られます。オペレーティング・システムは，現在実行中のルーチンからこれらのハンドラの検索を開始します。ルーチンにハンドラが設定されていない場合，呼び出しスタックの次のルーチンによって設定されたハンドラを検索します。このようにして，必要ならばメイン・プログラムまで検索します。

呼び出しフレーム・ハンドラの起動後，ハンドラは次のうち 1 つの処理を行います。

- 例外を処理し，プログラムの実行を続ける。
- 例外を再シグナル通知する。オペレーティング・システムが呼び出しスタックの次のハンドラを検索する。

- ブレークポイントまたはウォッチポイントを検出し，そのポイントで実行を一時停止する。
- 自分自身で例外を作成する。この場合， 1 次ハンドラが再び起動される。
- 終了する。プログラムの実行を終了する。

14.5.3.4 最終ハンドラおよびラスト・チャンス・ハンドラ

これらのハンドラはデバッガによって制御されます。これらのハンドラは，アプリケーションで宣言されたハンドラが例外を処理していなければ，デバッガに最終的に制御を戻し， DBG>プロンプトを表示します。そうでない場合は，デバッグ・セッションは終了し， DCL コマンド・インタプリタに制御が引き渡されます。

最終ハンドラは，呼び出しスタックの最終フレームであり，これら 2 つのハンドラのうち最初に起動されるハンドラです。次の例は，処理されていない例外が例外ブレークポイントから最終ハンドラに伝えられ場合，どのように処理されるかを示しています。

```
DBG> SET BREAK/EXCEPTION
DBG> GO
   .
   .
   .
%SYSTEM-F-INTDIV, arithmetic trap, integer divide by zero at PC=0000066C, PSL=03C00022
break on exception preceding TEST\%LINE 13
     6:         X := 3/Y;
DBG> GO
%SYSTEM-F-INTDIV, arithmetic trap, integer divide by zero at PC=0000066C, PSL=03C00022
DBG>
```

この例では，最初の INTDIV メッセージは 1 次ハンドラによって発行されます。2 つ目のメッセージは最終ハンドラによって発行され，その後最終ハンドラにより DBG>プロンプトが表示されています。

ラスト・チャンス・ハンドラは，呼び出しスタックの破損のために最終ハンドラが制御を得ることができなかった場合にだけ起動されます。次に例を示します。

```
DBG> DEPOSIT %FP = 10
DBG> GO
   .
   .
   .
%SYSTEM-F-ACCVIO, access violation, reason mask=00, virtual address=0000000A, PC=0000319C, PSL=03C00000
%DEBUG-E-LASTCHANCE, stack exception handlers lost, re-initializing stack
DBG>
```

キャッチオール・ハンドラはオペレーティング・システムの一部であり，ラスト・チャンス・ハンドラが制御を得ることができなかった場合に起動されます。キャッチオール・ハンドラはレジスタのダンプを作成します。デバッガがユーザ・プログラムを制御している場合に起動されることはありません。デバッガを使用せずにプログラムを実行しているとき，プログラムでエラーが検出されると起動されます。

デバッグ・セッションの間，レジスタ・ダンプが現れ DCL レベル($)に戻った場合は，弊社のサポート要員にご連絡ください。

14.5.4 例外関連の組み込みシンボル

例外がシグナル通知された場合，デバッガは次の例外関連の組み込みシンボルを設定します。

シンボル	説明
%EXC_FACILITY	現在の例外を発行したファシリティの名前
%EXC_NAME	現在の例外の名前
%ADAEXC_NAME	現在の Ada 例外の名前 (Ada プログラムの場合のみ)
%EXC_NUMBER	現在の例外の番号
%EXC_SEVERITY	現在の例外の重大度コード

これらのシンボルを次のように使用することができます。

- 現在の例外の条件コードのフィールドに関する情報を取得する。
- 例外ブレークポイントまたは例外トレースポイントを修飾し，ある種の例外でだけ検出されるようにする。

次に，これらのシンボルの使用例を示します。 WHEN 句の条件式は言語固有ですので注意してください。

```
DBG> EVALUATE %EXC_NAME
'ACCVIO'
DBG> SET TRACE/EXCEPTION WHEN (%EXC_NAME = "ACCVIO")
DBG> EVALUATE %EXC_FACILITY
'SYSTEM'
DBG> EVALUATE %EXC_NUMBER
12
DBG> EVALUATE/CONDITION_VALUE %EXC_NUMBER
%SYSTEM-F-ACCVIO, access violation, reason mask=01, virtual address=FFFFFF30, PC=00007552, PSL=03C00000
DBG> SET BREAK/EXCEPTION WHEN (%EXC_NUMBER = 12)
DBG> SET BREAK/EXCEPTION WHEN (%EXC_SEVERITY .NE. "I" .AND. %EXC_SEVERITY .NE. "S")
```

14.6 終了ハンドラのデバッグ

終了ハンドラは，イメージがシステム・サービス$EXIT を要求したり，実行が終了したりしたときに呼び出されるプロシージャです。ユーザ・プログラムは 1 つまたは複数の終了ハンドラを宣言することができます。デバッガは常に自分自身の終了ハンドラを宣言します。

プログラム終了時，アプリケーションで宣言された終了ハンドラがすべて実行されたあとにデバッガの終了ハンドラが実行されます。

アプリケーションで宣言された終了ハンドラをデバッグするには，次の手順に従ってください。

1. デバッグする終了ハンドラにブレークポイントを設定する。
2. 次のいずれか 1 つの方法で終了ハンドラを実行する。
   - 終了ハンドラを起動する命令をプログラム内に指定する。通常は$EXIT を呼び出す。
   - プログラムを終了させる。
   - EXIT コマンドを入力する。 QUIT コマンドはユーザが宣言した終了ハンドラを実行しないので注意する。

   終了ハンドラが実行されると，ブレークポイントが有効になり，制御がデバッガに戻る。続いてコマンドの入力を要求するプロンプトが表示される。

SHOW EXIT_HANDLERS コマンドは，ユーザ・プログラムで宣言した終了ハンドラを表示します。終了ハンドラのルーチンは，呼び出された順番に表示されます。ルーチン名は可能であればシンボルとして表示されます。そうでない場合はアドレスが表示されます。デバッガの終了ハンドラは表示されません。次に例を示します。

```
DBG> SHOW EXIT_HANDLERS
exit handler at STACKS\CLEANUP
exit handler at BLIHANDLER\HANDLER1
DBG>
```

14.7 AST ドライブ式プログラムのデバッグ

プログラムは，非同期システム・トラップ (AST) を明示的に使用したり，システムサービス，またはアプリケーションで定義された AST ルーチンを呼び出す実行時ライブラリ (RTL) ルーチンを呼び出すことによって，暗黙に使用したりできます。第 14.7.1 項では，ユーザのプログラムから発行された AST の実行要求を禁止したり許可にしたりすることによってデバッグを行う方法を説明します。

14.7.1 AST の実行要求の禁止と許可

AST ドライブ式プログラムのデバッグは混乱することがあります。デバッガが実行を続けている間 (コマンドを処理したり，例外をトレースしたり，情報を表示したりしている間) に，デバッグ中のプログラムからの割り込みが発生し，処理されないことがあるからです。

省略時の設定では， AST の実行要求はプログラムが実行されている間は許可されています。 DISABLE AST コマンドはプログラムが実行されている間の AST の実行要求を禁止し，このような割り込みの発生をキューに登録します。

AST の実行要求は，デバッガの実行中は常に禁止されます。

静的ウォッチポイントが有効な場合，デバッガは，システム・サービス呼び出しの直前に，静的ウォッチポイント， AST，スレッド切り替えをオフにします。デバッガは，システム・サービス呼び出しが終了した直後に再起動します。詳細については，SET WATCH コマンドの項を参照してください。 ENABLE AST コマンドは，待ち状態にある AST の実行要求も含め， AST の実行要求を再び許可します。 SHOW AST コマンドは， AST の実行要求が禁止されているか許可されているかを示します。

CALL コマンドで呼び出されたルーチンの実行中， AST の実行要求を制御するには，/[NO]ASC 修飾子を使用します。 CALL/AST コマンドは，呼び出されたルーチン内の AST の実行要求を許可します。 CALL/NOAST コマンドは，呼び出されたルーチン内の AST の実行要求を禁止します。 CALL コマンドで/AST も/NOAST も指定しない場合は，前に DISABLE AST コマンドを入力していないかぎり， AST の実行要求は許可されます。

14.8 変換されたイメージのデバッグ (Alpha および Integrity のみ)

OpenVMS Alpha システムおよび Integrity システムの場合，デバッガは，変換されたイメージのデバッグをサポートしてしません。変換されたイメージをデバッグする必要があるときは， Delta/XDelta デバッガを使用してください。このデバッガの詳細については，『 OpenVMS Delta/XDelta Debugger Manual』を参照してください。

14.9 同期化または通信機能を実行するプログラムのデバッグ

同期化または通信を実行する一部のプログラムでは，デバッグで問題が発生することがあります。たとえば，デバック中のアプリケーションで$DEQ システム・サービス呼び出しに LCK$M_DEQALL 修飾子が含まれている場合などです (この修飾子は，ユーザ・プロセス (カーネル) のデバッガの部分とデバッガのメイン・プロセスとの間の通信リンクを破壊します)。

14.10 インライン・ルーチンのデバッグ

OpenVMS システムでは，デバッガはインライン・ルーチンのデバッグをサポートしません。インライン・ルーチンをデバッグしようとすると，次の例に示すように，デバッガはルーチンにアクセスできないことを示すメッセージを出力します。

```
%DEBUG-E-ACCESSR, no read access to address 00000000
```

この問題を回避するには，/NOOPTIMIZE 修飾子を使用してプログラムをコンパイルします。

15
マルチプロセス・プログラムのデバッグ

本章では，デバッガの機能のうちマルチプロセス・プログラム (2 つ以上のプロセスで動作するプログラム) 独自のものについて説明します。これらの機能を使用して，プロセス情報を表示したり，特定のプロセスの実行を制御することができます。他章で説明した機能とあわせて使用してください。

本章で扱うすべてのイメージはデバッグ可能であり，デバッガで制御することができます。/NOTRACEBACK 修飾子を指定してリンクされたイメージは，デバッガの制御下に置くことはできません。第 1.2 節で説明したように，イメージのデバッグ時にすべてのシンボル情報にアクセスできるのは，そのモジュールを/DEBUG 修飾子を使用してコンパイルおよびリンクするときだけです。

15.1 基本的なマルチプロセス・デバッグ方法

この節では，マルチプロセス・デバッグの基本的な概念を紹介します。詳しい情報については，後の節を参照してください。

15.1.1 マルチプロセス・デバッグ・セッションの開始

この項では，マルチプロセス・デバッグ・セッションを開始する最も簡単な方法について説明します。第 15.16.3 項では，デバッガのその他の起動方法について説明しています。

マルチプロセス・デバッグ・セッションを開始するには，保持デバッガを起動します。次に例を示します。

```
$ debug/keep
  OpenVMS I64 Debug64 Version T8.2-008
DBG>
```

マルチプロセス・デバッグ・セッションでは，デバッガは制御下に置かれた個々の新しいプロセスをトレースします。デバッガは， Example 15–1 のような 10 進数のプロセス番号を使って，各プロセスを識別します。

Example 15–1 RUN/NEW コマンド

```
DBG> SHOW PROCESS
 Number  Name          State           Current PC
*    1 DBGK$$2727282C  activated       SERVER\__main
DBG> RUN/NEW CLIENT
process 2
  %DEBUG-I-INITIAL, Language: C, Module: CLIENT
  %DEBUG-I-NOTATMAIN, Type GO to reach MAIN program
  predefined trace on activation at CLIENT\__main
all> SHOW PROCESS
 Number  Name          State           Current PC
*    1 DBGK$$2727282C  activated       SERVER\__main
     2 USER_2          activated       CLIENT\__main
all>
```

Example 15–1 の RUN/NEW CLIENT コマンドは，新しいプロセスの中でプログラム CLIENT を起動します。デバッガは，(そのデバッグ・セッションの中で) 初めて複数のプロセスが制御下に置かれた時点で，制御下に置かれているすべてのプロセスのセットを識別するためにプロンプトをall>に変更します。

プロセスとプロセス・セット

デバッガが複数のプロセスを認識すると，デバッガ・プロンプトは現在のプロセス・セットの識別子の後に右山括弧 (>) を続けたものに変更されます。

概念上，各プロセスは，省略時にはデバッガがそのプロセスを制御下に置いたときに割り当てられた一意の 10 進数によって識別される，1 つのプロセスが含まれるセットに属しています。プロセスは複数のセットに属することができます。デバッガの制御下に置かれたすべてのプロセスは，省略時にはallという名前のセットにグループ化されます。

DEFINE /PROCESS_SETコマンドを使用すると，ユーザが名前を付けたセットにプロセスをグループ化することができます。

現在のプロセス・セット

デバッガ・コマンドは，省略時には**現在のプロセス・セット**に適用されます。省略時の設定では，現在のプロセス・セットは all という名前のセットです。SET PROCESSコマンドを使用すると，現在のプロセス・セットを変更することができます。

コマンド・プロセス・セット

コマンドの対象となるプロセスのセットは**コマンド・プロセス・セット**と呼ばれます。省略時のコマンド・プロセス・セットは，現在のプロセス・セットです。

プロセス・セット接頭辞

現在のプロセス・セットを変更することなく，現在のプロセス・セット以外のコマンド・プロセス・セットにデバッガ・コマンドを適用することができます。このためには，コマンドの接頭辞として，プロセス・セットの名前の後に右山括弧 (>) を続けます。次に例を示します。

```
all> 1,2,5> GO
```

1,2,5>は**プロセス・セット接頭辞**です。この構文により，前のコマンド行からコマンドをカットしてペーストすることができます。

可視プロセス

可視プロセスは現在のディスプレイに表示されているプロセスのことで，SHOW PROCESSディスプレイの最左端の欄にアスタリスク (*) が付いています。可視プロセスはSET PROCESS/VISIBLEコマンドで変更できます。次に例を示します。

```
all> SHOW PROCESS
 Number  Name          State           Current PC
*    1 DBGK$$2727282C  activated       SERVER\__main
     2 USER_2          activated       CLIENT\__main
all>
```

上の例では，プロセス番号 1 が可視プロセスです。

15.2 プロセス情報の取得

現在のデバッグ・セッションの制御下にあるプロセス情報を取得するには， SHOW PROCESS コマンドを使用します。省略時の設定では， SHOW PROCESS コマンドは，デバッガ制御下のすべてのプロセス情報を表示します。(これらはプロセス・セットの全プロセスです。) Example 15–2 は，デバッガ起動直後に表示される情報の 1 例です。

Example 15–2 SHOW PROCESS **コマンド**

```
DBG> SHOW PROCESS/BRIEF/ALL
 Number  Name               State           Current PC
*    1 JONES               activated       MAIN_PROG\%LINE 2
DBG>
```

修飾子/BRIEF と/ALL は省略時の設定です。また，デバッガはまだ 1 つのプロセスしか制御下に置いていないので，省略時のプロンプトを表示していることに注意してください。 SHOW PROCESS コマンドは，指定された各プロセスについて以下の情報を表示します。

- デバッガによって割り当てられたプロセス番号。 Example 15–2 では，デバッガが最初に認識したプロセスなので，プロセス番号は 1 になる。最左端の欄のアスタリスク(*)は，可視プロセスを示す。
- プロセス名。この例では，プロセス名は JONES である。
- そのプロセスの現在のデバッグ状態。プロセスは，最初にデバッガ制御下に置かれたとき (デバッガの制御下でプログラムを実行する直前)，起動状態にある。表 15–1 は，デバッガ制御下のプロセスのデバッグ状態の取りうる値を要約したものである。

- そのプロセス内でイメージの実行が停止している記憶位置 (可能ならシンボル化されたもの)。 Example 15–2 では，イメージはまだ実行を開始していない。

表 15–1 デバッグ状態

状態	説明
Running	デバッガ制御下での実行。
Stopped	
Activated	イメージとそのプロセスがデバッガの制御下に置かれた直後。
Break[1]	ブレークポイントが検出された。
Interrupted	次の方法の 1 つで，プロセスの実行が中断された。 • 他プロセスで実行が中断された。 • 強制終了キー・シーケンス (省略値の設定では Ctrl/C) で中断された。 • STOP コマンドによって中断された。
Step[1]	STEP コマンドが終了した。
Trace[1]	トレースポイントが検出された。
Unhandled exception	未処理例外が発生した。
Watch of	ウォッチポイントが検出された。
Terminated	イメージの実行は終了したが，プロセスはまだデバッガの制御下に置かれている。したがって，ユーザは，イメージとそのプロセスに関する情報を得ることができる。

[1] 上記以外のデバッグ状態については， SHOW PROCESS コマンドの説明を参照してください。

Example 15–2 で， SHOW PROCESS コマンドの後ろに STEP コマンドを入力すると， SHOW PROCESS 表示の状態欄は，ステップ実行完了後，実行が停止していることを示します。次に例を示します。

```
DBG> STEP
DBG> SHOW PROCESS
 Number  Name                State           Current PC
*    1 JONES               step            MAIN_PROG\%LINE 3
DBG>
```

同じように，ブレークポイントを設定した後に GO コマンドを入力した場合，ブレークポイントがトリガされた後に SHOW PROCESS コマンドを入力すると，状態は break と表示されます。

15.3 プロセス指定

デバッガが接続する個々の新規プロセスは，**プロセス番号**によって識別されます。最初のプロセスにはプロセス番号 1，次のプロセスにはプロセス番号 2 が割り当てられます。プロセスが停止すると，そのプロセス番号はリサイクルされ，デバッガはそれ以降のプロセスに同じ番号を割り当てられるようになります。

プロセスは**プロセス指定**を使って参照されます。最も単純なプロセス指定は，プロセスの作成時に OpenVMS が作成するプロセス名と，デバッガがプロセスを制御下に置くときに作成するプロセス番号です。番号だけから構成されるプロセス指定はプロセス番号として解釈されます。デバッガ・コマンドの中では，プロセス番号を使って個々のプロセスを指定することができます (例: "2,3,4,5")。

プロセス指定項目としては名前を使用することができ，その場合にはプロセス名またはプロセス・セット名を参照することができます。デバッガはまずその名前を持つプロセス・セットを探します。これに失敗した場合，デバッガはその名前を持つプロセスを探します。%PROCESS_NAME レキシカル関数を使用すると，明示的にプロセス名を指定することができます。

Example 15–3 は，完全なプロセス指定構文を示しています。

Example 15–3 **プロセス指定構文**

```
process-spec ::=   process-spec-item [, process-spec-item]

process-spec-item ::= named-item |
                      numbered-item |
                      pid-item |
                      process-set-name |
                      special-item

named-item ::=     [%PROCESS_NAME] wildcard-name

numbered-item ::=  numbered-process

numbered-process ::= [%PROCESS_NUMBER] decimal-number

pid-item  ::=      %PROCESS_ID VMS-process-identifier

process-set-name ::= name

special-item ::=   %NEXT_PROCESS |
                   %PREVIOUS_PROCESS |
                   %VISIBLE_PROCESS
```

15.4 プロセス・セット

DEFINE PROCESS_SET コマンドの後に，コンマ (,) で区切ったプロセスのリストを指定することで，プロセスを**プロセス・セット**と呼ばれるグループに入れることができます。次に例を示します。

```
all> DEFINE/PROCESS CLIENTS = 2,3
all> SET PROCESS CLIENTS
clients> STEP
process 2,3
  stepped to CLIENT\main\%LINE 18796
   18796:     status = sys$crembx (0, &mbxchan, 0, 0, 0,
                                   0, &mbxname_dsc, CMB$M_READONLY, 0);
clients> SHOW PROCESS CLIENTS
 Number  Name          State           Current PC
     2 USER1_2         step     CLIENT\main\%LINE 18796
     3 USER1_3         step     CLIENT\main\%LINE 18796
clients>
```

all という名前の定義済みのプロセス・セットは，デバッガが初めて起動されたときの省略時のプロセス・セットとなっています。このプロセス・セットを再定義することはできません。

現在のプロセス・セット

デバッグ・セッションの中では，つねに現在のプロセス・セットが有効となっています。現在のプロセス・セットは，デバッガのプロセス依存コマンドが省略時の設定として適用されるプロセスのグループです。プロセスに依存するデバッガ・コマンドの一覧については，第 15.6 節を参照してください。

省略時の設定では，現在のプロセス・セットはすべてのプロセスのセットで，プロセス・セット名は all となっています。現在のプロセス・セットは SET PROCESS コマンドで変更できます。

SET PROCESS コマンドは 3 つの機能を持っています。

- 現在のプロセス・セットを指定する。
- /VISIBLE 修飾子で可視プロセスを制御する。
- /[NO]DYNAMIC 修飾子で動的なプロセス設定のオン/オフを切り替える。

修飾子なしで使用した場合， SET PROCESS コマンドは，現在のプロセス・セットを指定するプロセス指定を単一パラメータとして取ります。次に例を示します。

```
all> SET PROCESS 1
1> STEP
process 1
  stepped to SERVER\main\%LINE 18800
   18800:     if (!(status & 1))
1> SET PROCESS ALL
all>
```

SET PROCESS/DYNAMIC コマンドは， STEP コマンドの完了やブレークポイントのトリガといったデバッガ・イベントが発生したときに，可視プロセスを変更するようにデバッガに指示します。イベントをトリガしたプロセスが可視プロセスになります。次に例を示します。

```
all> SET PROCESS/DYNAMIC
all> 1> STEP
process 1
  stepped to SERVER\main\%LINE 18808
   18808:     df_p = fopen (datafile, "r+");
all> SHOW PROCESS/VISIBLE
 Number  Name          State            Current PC
*    1 DBGK$$2727282C  step     SERVER\main\%LINE 18808
all>
```

コマンド・プロセス・セット

コマンド・プロセス・セットは，デバッガ・コマンドが適用されるプロセスのグループです。省略時の設定では，コマンド・プロセス・セットは現在のプロセス・セットです。プロセス・セット接頭辞を使用して，現在のコマンドの対象となるコマンド・プロセス・セットを指定することができます。この場合には，その 1 つのコマンドについてのみ，現在のプロセス・セットが上書きされます。次に例を示します。

```
all> 2,3> STEP
processes 2,3
  stepped to CLIENT\main\%LINE 18797
   18797:     if (!(status & 1))
all> clients> STEP
processes 2,3
  stepped to CLIENT\main\%LINE 18805
   18805:     memset (&myiosb, 0, sizeof(myiosb));
all>
```

プロセス独立コマンドは，現在のプロセス・セットと同様に，プロセス・セット接頭辞をすべて無視します。

15.5 デバッガ・プロンプト

省略時の設定では，デバッガ・コマンド・プロンプトは，プロセス指定と同じ構文を使用して，現在のプロセス・セットを示しています。コマンド・プロンプトは，現在のプロセス・セットのプロセス指定の後に右山括弧 (>) を続けたものです。現在のプロセス・セットを定義すると，デバッガは現在のプロセス・セットの名前の後に右山括弧を続けたものにプロンプトを変更します。次に例を示します。

```
all>        ! by default, current process set is all processes
all>
all> SET PROCESS 2,3,4,5
2,3,4,5> DEFINE /PROCESS_SET interesting 1,2,3,7
2,3,4,5> SET PROCESS interesting
interesting> SET PROCESS *
all> SET PROCESS 3
3>
```

注意

デバッガは複数のプロセスを認識するまで，プロセス指定形式のデバッガ・プロンプトを使用しません。

15.6 プロセス依存コマンド

コマンドにはプロセス依存コマンドとプロセス独立コマンドの 2 つの種類があります。

プロセス依存コマンドは， GO， STEP， CALL，および SET BREAK のように，プロセスの状態に依存するコマンドです。

プロセス独立コマンドは， SET DISPLAY， WAIT， ATTACH，および SPAWN のように，デバッガの状態に依存し，またデバッガの状態に影響を与え，プロセスの状態は無視するコマンドです。

15.7 可視プロセスとプロセス依存コマンド

可視プロセスは，省略時の設定でソース・ディスプレイ (およびその他のプロセス指向のディスプレイ) に表示されるプロセスです。 SET PROCESS コマンドで現在のプロセス・セットが変更されると，可視プロセスはそのコマンドで指定された最初のプロセスに設定されます。 SET PROCESS/VISIBLE コマンドを使用すると，現在のプロセス・セットを変更せずに，特定のプロセスを可視プロセスとして指定することができます。

15.8 プロセス実行の制御

複数のプロセスを持つアプリケーションをデバッグするときには，他のプロセスの実行中に特定のプロセスを停止するということがよく行われます。すべてのプロセスが停止するのを待たずに，停止されたプロセスに対してのみコマンドを指定できると便利です。このような機能は WAIT モードによって実現されています。

15.8.1 WAIT モード

プロセスの実行に関しては，デバッガは WAIT モードと NOWAIT モードの 2 つのモードを持っています。SET MODE [NO]WAITコマンドを使って WAIT モードを切り替えることで，デバッガが新たなコマンドを受け付けて実行するまでに，すべての実行中のプロセスが停止するのを待つかどうかを制御することができます。省略時の設定は WAIT モードです。

デバッガが WAIT モードになっているときに GO， STEP，または CALL コマンドを入力すると，デバッガはコマンド・プロセス・セットに含まれるすべてのプロセスでそのコマンドを実行し，これらのプロセスが (たとえばブレークポイントで) すべて停止するまで待ってから，プロンプトを表示し，新たなコマンドを受け付けます。

デバッガが NOWAIT モードになっているときに GO， STEP，または CALL コマンドを入力すると，デバッガはコマンド・プロセス・セットに含まれるすべてのプロセスでそのコマンドを実行し，ただちにプロンプトを表示します。この際には，すべてのプロセスが停止しているかどうかにかかわらず，ただちに新しいコマンドを入力することができます。この機能は，特にマルチプロセス・プログラムをデバッグしているときには，高度な柔軟性を提供してくれます。

WAIT モードを制御することによって，以下のような操作が可能となります。

- プログラムの実行中に，デバッガをソース・ブラウザとして使用できる。ソース・ビューはプロセス独立なので，プロセスの実行中にそのフォーカスを変更することができる。
- 各プロセスを個別に制御することができる。
- 複数のプロセスを一度に制御することができる。

SET MODE [NO]WAIT コマンドは，次の SET MODE [NO]WAIT コマンドが入力されるまで有効となります。次に例を示します。

```
all> SET MODE NOWAIT
all> clients> STEP
all> SHOW PROCESS
 Number  Name          State            Current PC
     1 DBGK$$2727282C  step     SERVER\main\%LINE 18819
     2 USER1_2         running          not available
*    3 USER1_3         running          not available
all>
```

WAIT コマンドを使用して， 1 つのコマンドの間だけ NOWAIT モードを無効にし，コマンド・プロセス・セット内のすべてのプロセスが停止するのを待ってから新たなコマンドを受け付けるようデバッガに指示することができます。このコマンドが完了すると， NOWAIT モードが有効となります。次に例を示します。

```
all> GO;WAIT
processes 2,3
  break at CLIENT\main\%LINE 18814
        18814:       status = sys$qiow (EFN$C_ENF,  mbxchan,
IO$_READVBLK|IO$M_WRITERCHECK, &myiosb,
process 1
  break at SERVER\main\%LINE 18834
        18834:       if ((myiosb.iosb$w_status ==
                          SS$_NOREADER) && (pos_status != -1))
all>
```

コマンドが非対話的に処理されるとき (FOR， REPEAT， WHILE， IF，および@コマンドの中や， WHEN 句の中のデバッガ・コマンド・シーケンス)，省略時の設定で，コマンド・シーケンスの中では WAIT モードが有効となります。

NOWAIT モードでは， EXAMINE コマンドは (すべてのプロセス独立コマンドと同様に) コマンド・プロセス・セットの中の停止されたプロセスの結果を表示します。コマンド・プロセス・セットの中のすべてのプロセスが実行中だった場合，EXAMINE コマンドはその旨を報告し，デバッガはプロンプトを表示して，新しいコマンドを受け付けるようになります。同じように， NOWAIT モードでの GO コマンドは，コマンド・プロセス・セットの中のすべての停止中のプロセスを開始します。

15.8.2 割り込みモード

SET MODE [NO]INTERRUPTコマンドを使用すると，割り込みモードの状態を切り替えることができます。割り込みモードがオンになっていると，デバッガは 1 つのプロセスが停止した時点ですべてのプロセスを停止します。割り込まれたプロセスが RTL やシステム・サービス呼び出しの深い位置にある場合には，プロセス・スタック上に無意味な非シンボリック・フレームが多数残るため，不便になることがあります。

割り込みモードがオフになっていると，デバッガは STOP コマンドが入力されない限り，他のプロセスは停止しません。これは省略時のモードです。

15.8.3 STOP コマンド

STOP コマンドを使用すると，実行中のプロセスに割り込むことができます。 STOP コマンドは，コマンド・プロセス・セットの中のすべての実行中のプロセスに割り込みます。

STOP コマンドは，コマンド・セットの中のすべての実行中のプロセスに停止要求を送信した時点で完了します。次に例を示します。

```
all> SHOW PROCESS
 Number  Name          State            Current PC
     1 DBGK$$2727282C  break    SERVER\main\%LINE 18834
     2 USER1_2         running          not available
*    3 USER1_3         running          not available
all> clients> STOP
all> SHOW PROCESS
 Number  Name          State            Current PC
     1 DBGK$$2727282C  break    SERVER\main\%LINE 18834
     2 USER1_2         interrupted   0FFFFFFFF800F7A20
*    3 USER1_3         interrupted   0FFFFFFFF800F7A20
all>
```

15.9 他のプログラムへの接続

保持デバッガ・セッションから，デバッグ可能なプログラムをデバッガの制御下に置くことができます。これは，他のプロセス内で独立に実行されているクライアント・プログラムであってもかまいません。デバッガはまだそのプロセスを認識していないため， SHOW PROCESS コマンドでプロセスに関する情報を得ることはできません。 CONNECT コマンドを入力し，デバッガの%PROCESS_NAME レキシカル関数を使用して，クライアント・プログラムのプロセス名を指定してください。次に例を示します。

```
all> CONNECT %PROCESS_NAME CLIENT2
process 3
  predefined trace on activation at 0FFFFFFFF800F7A20
all> SHOW PROCESS
 Number  Name          State            Current PC
*    1 DBGK$$2727282C  activated        SERVER\__main
     2 USER1_2         activated        CLIENT\__main
     3 CLIENT2         interrupted   0FFFFFFFF800F7A20
                        activated
all>
```

デバッガのメイン・プロセスとクライアントが実行されているプロセスの間で，いずれかのデバッガ論理名 (DEBUG， DEBUGSHR， DEBUGISHR， DBGTBKMSG， DBG$HELP， DBG$UIHELP， DEBUGAPPCLASS，および VMSDEBUGUIL) が異なっている場合， CONNECT コマンドを入力すると予期しない結果が生じることがあります。

15.10 スポーンされたプロセスへの接続

保持デバッガでデバッグ中のプログラムがデバッグ可能なプロセスをスポーンしたとき，スポーンされたプロセスはデバッガに未接続の状態のまま待機しています。この時点で，デバッガは新しくスポーンされたプロセスに関する情報を持っておらず， SHOW PROCESS コマンドでそのプロセスに関する情報を得ることはできません。新しくスポーンされたプロセスをデバッガの制御下に置くには，次のいずれかの方法を使用します。

- 実行を開始する STEP などのコマンドを入力する (以下の例のようにプログラムが階層的なモデルに従っている場合)。
- パラメータを指定せずに CONNECT コマンドを入力する。 CONNECT コマンドは，プロセスをそれ以上実行したくないときに有効である。

次に， CONNECT コマンドの使用時の例を示します。

```
1> STEP
stepped to MAIN_PROG\%LINE 18 in %PROCESS_NUMBER 1
18:        LIB$SPAWN("RUN/DEBUG TEST",,,1)
1> STEP
stepped to MAIN_PROG\%LINE 21 in %PROCESS_NUMBER 1
21:         X = 7
1> CONNECT
predefined trace on activation at routine TEST in %PROCESS_NUMBER 2
all>
```

この例の第 2 の STEP コマンドは，プロセスをスポーンする LIB$SPAWN 呼び出しを通り過ぎます。 CONNECT コマンドは待機中のプロセスをデバッガの制御下に置きます。 CONNECT コマンドを入力したときには，プロセスが接続されるまでしばらく待たなければならないことがあります。"predefined trace on . . . "というメッセージは，デバッガがプロセス 2 として識別された新しいプロセスを制御下に置いたことを示しています。

この時点で SHOW PROCESS コマンドを入力すると，各プロセスのデバッグ状態と，実行が停止されている位置が表示されます。

```
all> SHOW PROCESS
 Number  Name            State          Current PC
*    1 JONES             step           MAIN_PROG\%LINE 21
     2 JONES_1           activated      TEST\%LINE 1+2
all>
```

CONNECT コマンドは，デバッガへの接続を待っているすべてのプロセスをデバッガの制御下に置くことに注意してください。待機中のプロセスがない場合は， Ctrl/C を押して CONNECT コマンドを強制終了し，デバッガ・プロンプトを表示させることができます。

デバッガ・プロセスとスポーンされたプロセスの間で，いずれかのデバッガ論理名 (DEBUG， DEBUGSHR， DEBUGISHR， DBGTBKMSG， DBG$HELP， DBG$UIHELP， DEBUGAPPCLASS，および VMSDEBUGUIL) が異なっている場合， CONNECT コマンドを入力すると予期しない結果が生じることがあります。

15.11 イメージの終了のモニタ

プロセスのメイン・イメージが最後まで実行されて完了すると，プロセスは終了デバッグ状態に移行します (オペレーティング・システムのレベルでのプロセス終了と混同してはなりません)。この条件は，省略時の設定で， SET TRACE/TERMINATING コマンドが入力されたかのようにトレースの対象となっています。

デバッガは終了デバッグ状態になったプロセスも認識しており， SHOW PROCESS ディスプレイに表示します。このため，変数を確認するコマンドなどを入力することができます。

15.12 デバッガの制御からのプロセスの解放

プロセスを終了せずにそのプロセスをデバッガの制御から解放するには，DISCONNECT コマンドを入力します。一方，プロセスに EXIT または QUIT コマンドを指定すると，そのプロセスは終了します。

DISCONNECT コマンドは，クライアント／サーバ・モデルのプログラムには必須です。次に例を示します。

```
all> SHOW PROCESS
 Number  Name          State           Current PC
*    1 DBGK$$2727282C  step     SERVER\main\%LINE 18823
     2 USER1_2         step     CLIENT\main\%LINE 18805
     3 USER1_3         step     CLIENT\main\%LINE 18805
all> DISCONNECT 3
all> SHOW PROCESS
 Number  Name          State           Current PC
*    1 DBGK$$2727282C  step     SERVER\main\%LINE 18823
     2 USER1_2         step     CLIENT\main\%LINE 18805
all> QUIT 1,2
DBG> SHOW PROCESS
  %DEBUG-W-NOPROCDEBUG, there are currently no processes being debugged
DBG> EXIT
$
```

デバッガ・カーネルは，デバッグされているイメージと同じプロセス内で実行されることに注意してください。このプロセスに対して DISCONNECT コマンドを発行すると，プロセスは解放されますが，カーネルはアクティブなままです。この状態はプログラム・イメージが実行を終了するまで続きます。1 つまたは複数の切断された，しかしアクティブなカーネルがユーザ・プログラム空間を占有しているときに，新しいバージョンのデバッガをインストールすると，そのプログラム・イメージへの再接続を試みたときにデバッガが異常な動作を見せることがあります。

15.13 特定のプロセスの終了

デバッグ・セッションを終了することなく，指定したプロセスだけを終了するには，EXIT または QUIT コマンドのパラメータとして 1 つまたは複数のプロセスを指定します。たとえば，

```
all> SHOW PROCESS
 Number  Name          State           Current PC
*    1 DBGK$$2727282C  step     SERVER\main\%LINE 18823
     2 USER1_2         step     CLIENT\main\%LINE 18805
all> QUIT 1,2
DBG> SHOW PROCESS
  %DEBUG-W-NOPROCDEBUG, there are currently no processes being debugged
DBG> EXIT
$
```

15.14 プログラム実行への割り込み

Ctrl/C，または SET ABORT_KEY コマンドによって設定された強制終了キー・シーケンスを押すことによって，現在イメージを実行しているすべてのプロセス内で，実行に割り込みをかけることができます。これは， SHOW PROCESS の表示では，Interrupted 状態として示されます。

Ctrl/C を使用してデバッガ・コマンドを強制終了させることもできます。

また，デバッガ STOP コマンドでプロセスを中止することもできます。

15.15 デバッグ・セッションの終了

すべてのデバッグ・セッションを終了するには，何もパラメータを指定しないで EXIT コマンドまたは QUIT コマンドを入力します。

EXIT コマンドは，プログラム内で宣言されている任意の終了ハンドラを実行します。 QUIT コマンドは，終了ハンドラを実行しません。

QUIT コマンド

QUIT コマンドは，実行中のプロセスを終了させます。 QUIT コマンドは，終了ハンドラを実行せずに，コマンド・プロセス・セット内のすべての実行中のプロセスを終了させます。 QUIT コマンドの前のプロセス・セット接頭辞は無視されます。次に例を示します。

```
all> SHOW PROCESS
 Number  Name          State           Current PC
*    1 DBGK$$2727282C  step     SERVER\main\%LINE 18823
     2 USER1_2         step     CLIENT\main\%LINE 18805
all> QUIT 1,2
DBG> SHOW PROCESS
  %DEBUG-W-NOPROCDEBUG, there are currently no processes being debugged
DBG> EXIT
$
```

QUIT コマンドは現在のプロセス・セットを無視します。プロセスを指定しなかった場合， QUIT コマンドはすべてのプロセスを終了した後に，デバッグ・セッションを終了します。

EXIT コマンド

EXIT コマンドは，実行中のプロセスを終了させます。 EXIT コマンドは，終了ハンドラを実行し，コマンド・プロセス・セット内のすべての実行中のプロセスを終了させます。 EXIT コマンドの前のプロセス・セット接頭辞は無視されます。次に例を示します。

```
all> SHOW PROCESS
 Number  Name          State            Current PC
*    1 DBGK$$2727282C  step     SERVER\main\%LINE 18823
     2 USER1_2         step     CLIENT\main\%LINE 18805
all> EXIT 1,2
DBG> SHOW PROCESS
  %DEBUG-W-NOPROCDEBUG, there are currently no processes being debugged
DBG> EXIT
$
```

EXIT コマンドは現在のプロセス・セットを無視します。プロセスを指定しなかった場合，EXIT コマンドはすべてのプロセスを終了した後に，デバッグ・セッションを終了します。

15.16 補足情報

この節では，第 15.1 節の内容よりもさらに高度な概念および使用法などの詳細な情報について説明します。

15.16.0.1 デバッグ時のプロセス関係

デバッガは，コードの大部分を含む**メイン・デバッガ**・イメージ (DEBUGSHR.EXE) と，より小さな**カーネル・デバッガ**・イメージ (DEBUG.EXE) の 2 つの部分からなります。このように分割されているため，デバッガとデバッグされるプログラムが干渉し合う可能性が少なくなり，マルチプロセス・デバッグ構成も可能となります。

保持デバッガの制御下にプログラムを置くと，メイン・デバッガはサブプロセスを作成して，プログラムとカーネル・デバッガを並行に動作させます。

デバッグ中のアプリケーションは，いくつかのプロセスで実行されます。デバッガ制御下で実行している各プロセスは，カーネル・デバッガのローカル・コピーが実行しています。自身のプロセス内で動作しているメイン・デバッガは，カーネル・デバッガを通して他のプロセスと通信します。

すべてのプロセスは，同じ UIC グループ内に存在しなければなりませんが，特定のプロセス/サブプロセス関係の中で関連づけられる必要はありません。また，異なるプロセス内で動作するプログラム・イメージが，お互いに通信する必要はありません。

マルチプロセス・デバッグに関するシステム要件については，第 15.16.6 項を参照してください。

15.16.1 デバッガ・コマンド内のプロセス指定

デバッガ・コマンド内でプロセスを指定する場合は，CONNECT コマンドを使用したプロセス指定を除けば，表 15–2 に示した形式のどれでも使用できます。第 15.9 節を参照してください。

CONNECT コマンドを使用すると，デバッガがまだ認識していないプロセスをデバッガの制御下に置くことができます。新しいプロセスは，デバッガの制御下に置かれるまではデバッガが割り当てたプロセス番号を持たず，組み込みプロセス・シンボル (%NEXT_PROCESS など) を使って参照することもできません。したがって，CONNECT コマンドにプロセスを指定するには，プロセス名またはプロセス識別子 (PID) のいずれかしか使用できません。

表 15–2 プロセス指定

形式	使用方法
[%PROCESS_NAME] *process-name*	スペースや小文字を含まないプロセス名。プロセス名にはワイルドカード文字(*)を含めることができる。
[%PROCESS_NAME] "*process-name*"	スペースまたは小文字を含むプロセス名。二重引用符(")の代わりに，一重引用符を (') 使用することもできる。
%PROCESS_PID *process_id*	プロセス識別子 (PID，16 進数)。
[%PROCESS_NUMBER] *process-number* (または%PROC *process-number*)	デバッガの制御下に入ったときにプロセスに割り当てられた番号。新しい番号は，1 から順番に各プロセスに割り当てられる。EXIT コマンドまたは QUIT コマンドによってプロセスが終了した場合，そのデバッグ・セッション中にその番号が再割り当てされることがある。プロセス番号は SHOW PROCESS コマンドを実行して表示できる。プロセスは，組み込みシンボル%PREVIOUS_PROCESS および%NEXT_PROCESS によってインデックスづけできるように，循環リスト内に順序づけされる。
process-set-name	DEFINE/PROCESS_SET コマンドで定義された，プロセスのグループを表すシンボル。
%NEXT_PROCESS	デバッガの循環プロセス・リスト中で可視プロセスの次のプロセス。
%PREVIOUS_PROCESS	デバッガの循環プロセス・リストの中で可視プロセスの前のプロセス。
%VISIBLE_PROCESS	シンボル，レジスタ値，ルーチン呼び出し，ブレークポイントなどの検索時に，スタック，レジスタ・セット，およびイメージが現在のコンテキストになっているプロセス。

コマンド入力の際は，次のように，組み込みシンボル%PROCESS_NAME および%PROCESS_NUMBER を省略することができます。たとえば，次のようになります。

```
2> SHOW PROCESS 2, JONES_3
```

組み込みシンボルの%VISIBLE_PROCESS，%NEXT_PROCESS，および%PREVIOUS_PROCESS は， IF， WHILE， REPEAT の各コマンドに基づいた制御構造内やコマンド・プロシージャ内で利用できます。

15.16.2 プロセスの起動と終了のモニタ

省略時の設定では，トレースポイントは，プロセスがデバッガの制御下に置かれたとき，およびプロセスがイメージ終了命令を実行したときに検出されます。これらの定義済みトレースポイントは， SET TRACE/ACTIVATING コマンドおよび SET TRACE/TERMINATING コマンドをそれぞれ入力した場合と同じ結果をもたらします。 SET BREAK/ACTIVATING コマンドおよび SET BREAK/TERMINATING コマンドによって，これらのイベント発生時にブレークポイントを設定することができます。

定義済みのトレースポイントを取り消すには， CANCEL TRACE/PREDEFINED コマンドを/ACTIVATING 修飾子および/TERMINATING 修飾子といっしょに使用します。ユーザ定義の起動時および終了時のブレークポイントを取り消すには，CANCEL BREAK コマンドを/ACTIVATING 修飾子および/TERMINATING 修飾子といっしょに使用します。/USER 修飾子は，ブレークポイントまたはトレースポイントを取り消す場合の省略時の設定です。

デバッガ・プロンプトは，最初のプロセスがデバッガの制御下に置かれたときに表示されます。その結果，単一プロセス・プログラムの場合のように，メイン・イメージの実行が開始される前にコマンドを入力できます。

同様に，デバッガ・プロンプトは，最後のプロセスがイメージ終了命令を実行したときにも表示されます。その結果，単一プロセス・プログラムの場合のように，プログラムの実行の終了後，コマンドを入力できます。

15.16.3 イメージの実行に割り込みをかけてデバッガに接続する方法

ユーザは，デバッガの制御外で動作しているあるプロセス内のデバッグ可能なイメージに割り込みをかけて，そのプロセスをデバッガに接続することができる。

- 新しくデバッグ・セッションを開始するには， DCL レベルで， Ctrl/Y–DEBUG シーケンスを使用する。この場合には，マルチプロセス・プログラムのデバッグには使用できない非保持デバッガが起動されることに注意すること。
- イメージに割り込みをかけて，既存のマルチプロセス・デバッグ・セッションに接続するには，デバッガ CONNECT コマンドを使用する。

15.16.4 マルチプロセス・デバッグの画面モード機能

省略時の設定では，ソース，命令，およびレジスタ・ディスプレイは，可視プロセスに関する情報を表示します。

/PROCESS 修飾子を DISPLAY コマンドといっしょに使用して，それぞれプロセス固有のディスプレイを作成したり，既存のディスプレイをプロセス固有にすることができます。プロセス固有のディスプレイは，そのプロセスのコンテキスト内で生成お

よび修正されます。ユーザは， PROMPT ディスプレイを除く任意のディスプレイをプロセス固有にすることができます。たとえば，次のコマンドは，プロセス 3 において，実行が中断している箇所のソース・コードを表示する，ソース・ディスプレイ SRC_3 を自動的に更新します。

```
2> DISPLAY/PROCESS=(3) SRC_3 AT RS23 -
2> SOURCE (EXAM/SOURCE .%SOURCE_SCOPE\%PC)
```

同じ方法で，プロセス固有のディスプレイに属性を割り当てて，プロセス固有でないディスプレイにすることもできます。たとえば，次のコマンドは， SRC_3 ディスプレイを，現在のスクロール・ディスプレイおよびソース・ディスプレイにします。すなわち， SCROLL， TYPE，および EXAMINE/SOURCE の各コマンドの出力は SRC_3 に変更されます。

```
2> SELECT/SCROLL/SOURCE SRC_3
```

プロセス指定なしの DISPLAY/PROCESS コマンドを入力すると，指定されたディスプレイが，そのコマンドが入力されたときの可視プロセスに固有のディスプレイとなります。たとえば，次のコマンドは，ディスプレイ OUT_X をプロセス 2 に固有のディスプレイにします。

```
2> DISPLAY/PROCESS OUT_X
```

マルチプロセス構成では，プロセス起動時の定義済みのトレースポイントが，新しくデバッガの制御下に置かれたプロセスごとに，新しいソース・ディスプレイと新しい機械語命令ディスプレイを自動的に作成します。これらのディスプレイ名は，それぞれソース・ディスプレイは SRC_*n*および機械語命令ディスプレイは INST_*n*という形式になります。ここで，*n*はプロセス番号です。これらのディスプレイは，最初に削除されたものとしてマークされます。それらは，プロセスが終了すると自動的に削除されます。

いくつかの定義済みキーパッド・キー・シーケンスを使用すれば，プロセスが起動されたときに自動的に作成されるプロセス固有のソース・ディスプレイおよび機械語命令ディスプレイを使用して，画面を構成できます。マルチプロセス・プログラム固有のキー・シーケンスは次のとおりです。 PF1 KP9， PF4 KP9， PF4 KP7， PF4 KP3， PF4 KP1。これらのシーケンスの意味については第 A.5 節を参照してください。これらのシーケンスに正確に対応するコマンドを知るには， SHOW KEY コマンドを使用します。

15.16.5 グローバル・セクション内でのウォッチポイントの設定 (Alpha および Integrity のみ)

Alpha および Integrity では，グローバル・セクション内にウォッチポイントを設定できます。グローバル・セクションは，マルチプロセス・プログラムのすべてのプロセス間で共有されるメモリ領域です。グローバル・セクション内のある記憶位置に設

定されたウォッチポイント (グローバル・セクション・ウォッチポイント) は，任意のプロセスがその記憶位置の内容を変更したときに検出されます。

SET WATCH コマンドを使用して配列やレコードにウォッチポイントを設定する場合には，構造体全体を指定するよりも各要素を指定したほうが性能が向上することに注意してください。

グローバル・セクションにマップされていない記憶位置にウォッチポイントを設定すると，そのウォッチポイントは従来の静的なウォッチポイントとして処理されます。次に例を示します。

```
1> SET WATCH ARR(1)
1> SHOW WATCH
watchpoint of PPL3\ARR(1)
```

この後， ARR がグローバル・セクションにマップされると，ウォッチポイントは自動的にグローバル・セクション・ウォッチポイントとみなされるようになり，それを通知する情報メッセージが発行されます。次に例を示します。

```
1> GO
%DEBUG-I-WATVARNOWGBL, watched variable PPL3\ARR(1) has
        been remapped to a global section
predefined trace on activation at routine PPL3 in %PROCESS_NUMBER 2
predefined trace on activation at routine PPL3 in %PROCESS_NUMBER 3
watch of PPL3\ARR(1) at PPL3\%LINE 93 in %PROCESS_NUMBER 2
    93:          ARR(1) = INDEX
   old value: 0
   new value: 1
break at PPL3\%LINE 94 in %PROCESS_NUMBER 2
   94:           ARR(I) = I
```

ウォッチされている記憶位置がグローバル・セクションにマップされると，そのウォッチポイントは，次のように，各プロセスから見えるようになります。

```
all> SHOW WATCH
For %PROCESS_NUMBER 1
  watchpoint of PPL3\ARR(1) [global-section watchpoint]
For %PROCESS_NUMBER 2
  watchpoint of PPL3\ARR(1) [global-section watchpoint]
For %PROCESS_NUMBER 3
  watchpoint of PPL3\ARR(1) [global-section watchpoint]
all>
```

15.16.6 デバッグのシステム要件

複数のユーザが同時にプログラムをデバッグすると，システムに負荷がかかります。本項では，デバッガが使用するリソースについて説明し，ユーザやシステム管理者が，プログラムのデバッグ作業用にシステムを調整できるようにします。

ここでは，デバッガが使用するリソースだけについて説明します。プログラム自体をサポートするには，システムを調整しなくてはならない場合もあります。

15.16.6.1 ユーザ・クォータ

各ユーザは，デバッガ用の追加プロセスを作成するために十分な PRCLM クォータを必要とします。このとき，プログラムの実行に必要な数以上のプロセスを作成できるようにします。

BYTLM， ENQLM， FILLM，および PGFLQUOTA は，プール・クォータです。これらのクォータは，デバッガ・プロセスを考慮して，次のように増やさなければならないに場合があります。

- 各ユーザの ENQLM クォータは，少なくともデバッグされるプロセス数の分だけ増やす。
- 各ユーザの PGFLQUOTA クォータも増やす。 PGFLQUOTA の値が不十分な場合，デバッガは起動できないか，または実行中に"virtual memory exceeded"のエラーを引き起こす。
- 各ユーザの BYTLM および FILLM クォータも増やす。デバッガは，デバッグされる各イメージ・ファイル，それらに対応するソース・ファイル，デバッガの入力，およびログ・ファイルをオープンするのに十分な BYTLM クォータおよび FILLM クォータを必要とする。

15.16.6.2 システム・リソース

カーネル・デバッガとメイン・デバッガはグローバル・セクションを通して通信を行います。個々のメイン・デバッガは，プラットフォームに関係なく，少なくとも 64 KB のグローバル・セクションを 1 つ使用します。 Alpha では，メイン・デバッガは，最大 6 個のカーネル・デバッガと通信できます。 Integrity では，メイン・デバッガが通信できるカーネル・デバッガの数は，最大 2 個までです。

15.17 例

Example 15–4 と Example 15–5 は，本章の例で使用しているサーバ・プログラムとクライアント・プログラムの C のコードを示しています。

Example 15–4 server.c

```
#include <stdio.h>
#include <starlet.h>
#include <cmbdef.h>
#include <types.h>
#include <descrip.h>
#include <efndef.h>
#include <iodef.h>
#include <iosbdef.h>
#include <ssdef.h>
#include <string.h>

#include "mbxtest.h"

int main (int argc, char **argv)
{
    unsigned int status, write_ef;
    char line_buf [LINE_MAX_LEN + 1];
    iosb myiosb;
    short mbxchan;

    /* Get event flag.  Look for or create the mailbox.
     */
    status = lib$get_ef (&write_ef);
    if (!(status & 1))
    {
 fprintf (stderr, "Server unable to get eventflag,
                                 status = %x", status);
 return 0;
    }
    status = sys$crembx (0, &mbxchan, 0, 0, 0, 0, &mbxname_dsc,
                                 CMB$M_WRITEONLY, 0);
    if (!(status & 1))
    {
 fprintf (stderr, "Server unable to open mailbox,
                                 status = %x", status);
 return 0;
    }

    /* Open for business.  Loop looking for and processing requests.
     */
    while (TRUE)
    {
 printf ("Input command: ");
 gets (&line_buf);
```

(次ページに続く)

Example 15–4 (続き) server.c

```
 status = sys$clref (write_ef);
 if (!(status & 1))
 {
     fprintf (stderr, "Client unable to clear read event flag,
                                  status = %x", status);
     return 0;
 }
 status = sys$qiow (write_ef, mbxchan,
                                  IO$_SETMODE | IO$M_READERWAIT, &myiosb,
                           0, 0, 0, 0, 0, 0, 0, 0);

 if ((status) && (myiosb.iosb$w_status))
 {
     status = sys$clref (write_ef);
     if (!(status & 1))
     {
  fprintf (stderr, "Client unable to clear read event flag,
                                  status = %x", status);
  return 0;
     }
     if (strlen (line_buf) == 0)
  status = sys$qio (write_ef, mbxchan, IO$_WRITEOF | IO$M_READERCHECK, &myiosb,
             0, 0, 0, 0, 0, 0, 0, 0);
     else
  status = sys$qio (write_ef, mbxchan, IO$_WRITEVBLK | IO$M_READERCHECK, &myiosb,
             0, 0, line_buf, strlen (line_buf), 0, 0, 0, 0);
     if (status)
     {
  status = sys$waitfr (write_ef);
  if ((myiosb.iosb$w_status & 1) && (status & 1))
                {
      if (strlen (line_buf) == 0)
   break;
  }
  else
      fprintf (stderr, "Server failure during write,
                             status = %x, iosb$w_status = %x\n",
               status, myiosb.iosb$w_status);
     }
     else
  fprintf (stderr, "Server failure for write request,
                             status = %x\n", status);
 }
 else
     fprintf (stderr, "Server failure during wait for reader,
                             status = %x, iosb$w_status = %x\n",
               status, myiosb.iosb$w_status);
    }
    printf ("\n\nServer done...exiting\n");
    return 1;
}
```

(次ページに続く)

Example 15–4 (続き) server.c

Example 15–5 client.c

```
#include <stdio.h>
#include <starlet.h>
#include <cmbdef.h>
#include <types.h>
#include <descrip.h>
#include <efndef.h>
#include <iodef.h>
#include <iosbdef.h>
#include <ssdef.h>
#include <string.h>

#include "mbxtest.h"

int main (int argc, char **argv)
{
    unsigned int status, read_ef;
    iosb myiosb;
    short mbxchan;
    char line_buf [LINE_MAX_LEN];

    /* Get event flag.  Look for or create the mailbox.
     */
    status = lib$get_ef (&read_ef);

    if (!(status & 1))
    {
 fprintf (stderr, "Client unable to get eventflag, status = %x", status);
 return 0;
    }
    status = sys$crembx (0, &mbxchan, 0, 0, 0, 0, &mbxname_dsc, CMB$M_READONLY, 0);
    if (!(status & 1))
    {
 fprintf (stderr, "Client unable to open mailbox, status = %x", status);
 return 0;
    }

    /* Loop requesting, receiving, and processing new data.
     */
    memset (&myiosb, 0, sizeof(myiosb));
```

(次ページに続く)

Example 15–5 (続き) client.c

```
    while (myiosb.iosb$w_status != SS$_ENDOFFILE)
    {
 status = sys$qiow (read_ef, mbxchan, IO$_SETMODE | IO$M_WRITERWAIT, &myiosb,
      0, 0, 0, 0, 0, 0, 0, 0);
 if ((status) && (myiosb.iosb$w_status))
 {
     status = sys$clref (read_ef);
     if (!(status & 1))
     {
  fprintf (stderr, "Client unable to clear read event flag, status = %x", status);
  return 0;
     }
     status = sys$qio (read_ef, mbxchan, IO$_READVBLK | IO$M_WRITERCHECK, &myiosb,
          0, 0, line_buf, sizeof(line_buf), 0, 0, 0, 0);
     if (status)
     {
  status = sys$waitfr (read_ef);
  if ((myiosb.iosb$w_status & 1) && (status & 1))
      puts (line_buf);
  else if ((myiosb.iosb$w_status != SS$_NOWRITER) &&
                            (myiosb.iosb$w_status != SS$_ENDOFFILE))
      fprintf (stderr, "Client failure during read,
                            status = %x, iosb$w_status = %x\n",
               status, myiosb.iosb$w_status);
     }
     else
  fprintf (stderr, "Client failure for read request, status = %x\n", status);
 }
 else
     fprintf (stderr, "Client failure during wait for writer,
                            status = %x, iosb$w_status = %x\n",
              status, myiosb.iosb$w_status);
 status = sys$clref (read_ef);
 if (!(status & 1))
 {
     fprintf (stderr, "Client unable to clear read event flag,
                            status = %x", status);
     return 0;
 }
    }
    printf ("\nClient done...exiting\n");
    return 1;
}
```

Example 15–4 および Example 15–5 に含まれるヘッダ・ファイル mbxtest.h は，以下のように表示されます。

```
$DESCRIPTOR(mbxname_dsc, "dbg$mptest_mbx");

#define LINE_MAX_LEN 255
```

16

タスキング・プログラムのデバッグ

本章では，タスキング・プログラム (マルチスレッド・プログラムとも呼ぶ) に固有のデバッガ機能について説明します。タスキング・プログラムは 1 つのプロセス内に複数のタスクまたは実行スレッドを持っています。デバッガで使用する**タスク**という用語は制御の流れのことであり，言語や実現方法とは関係ありません。デバッガのタスキング・サポートは，それらのプログラムすべてに適用されます。次のものを含んでいます。

- POSIX Threads か POSIX 1003.1b サービスを使用する言語で作成されたプログラム。これらのプログラムをデバッグするとき，デバッガの省略時のイベント機能は THREADS です。 Alpha と Integrity は POSIX Threads サービスを使用します。
- 言語固有のタスキング・サービス (言語が独自に用意しているサービス) を使用するプログラム。現在のところ，デバッガがサポートする組み込みタスキング・サービスを用意している言語は Ada だけです。 Ada プログラムをデバッグする場合は，デバッガの省略時のイベント機能は， ADA です。

注意

デバッガの中では，タスクとスレッドという用語は同義語として使われます。

PTHREAD$RTL **バージョン** 7.1 **またはそれ以降のバージョンがリンクされたプログラムをデバッグするときには，**PTHREAD **コマンドを使って** POSIX Threads **デバッガに直接アクセスすることができます。**

本章では， POSIX Threads 固有か言語固有の情報にはそのことを明記します。第 16.1 節に POSIX Threads 用語と Ada のタスキング用語の対応表を示します。

本章の機能を使用すれば，次のような処理を行うことができます。

- タスク情報を表示する。
- タスクの実行，優先順位，状態の遷移などを制御するためにタスク特性を変更する。
- タスク依存イベントと状態の遷移をモニタする。

これらの機能を使用するときには，同じタスキング・プログラムを実行してもそのときの状況によっては動作がデバッガにより変更されることがあるので注意してください。たとえば，現在アクティブなタスクの実行をあるブレークポイントで中断してい

るとき，入出力 (I/O) の終了による POSIX 信号か非同期システム・トラップ (AST) が届いた場合は，ユーザの続行指示後ただちにその他のタスクが適格になることがあります。

POSIX Threads についての詳しい説明は，『Guide to the POSIX Threads Library』を参照してください。 Ada タスクについての詳しい説明は Ada のマニュアルを参照してください。

マルチプロセス・プログラム (2 つ以上のプロセスに分けて実行されるプログラム) のデバッグについては第 15 章を参照してください。

16.1 POSIX Threads 用語と Ada 用語の対応表

表 16–1 に POSIX Threads と Ada の用語とその意味の対応を示します。

表 16–1 POSIX Threads 用語と Ada 用語の対応

POSIX Threads 用語	Ada 用語	意味
スレッド	タスク	同じプロセス内の制御の流れ
スレッド・オブジェクト	タスク・オブジェクト	制御の流れを表すデータ項目
オブジェクト名または式	タスク名または式	制御の流れを表すデータ項目
開始ルーチン	タスク本体	制御の流れに従って実行されるコード
該当なし	親タスク	親タスクの制御の流れ
該当なし	依存タスク	なんらかの親タスクに制御される子タスクの制御の流れ
同期化オブジェクト (ミューテクス，条件変数)	ランデブ構造 (エントリ呼び出しや accept 文など)	制御の流れを同期化する方法
スケジューリング方針およびスケジューリング優先順位	タスク優先順位	実行のスケジューリング方法
警告処理	abort 文	制御の流れの取り消し方法
スレッド状態	タスク状態	実行の状態 (待ち，レディ，実行中，終了)
スレッド作成属性 (優先順位，スケジューリング方針など)	プラグマ	パラレル・エンティティの属性

16.2 タスキング・プログラムの例

次の各項では，タスキング・プログラムのデバッグ時によく起こるエラーを含んでいるタスキング・プログラムの例を示します。

- 第 16.2.1 項では， POSIX Threads サービスを使用する C プログラムについて説明する。

- 第 16.2.2 項では，組み込みの Ada タスキング・サービスを使用する Ada プログラムについて説明する。

本章のその他の例は，これらのプログラムを引用したものです。

16.2.1 C のマルチスレッド・プログラムの例

Example 16–1 はマルチスレッドの C のプログラムです。条件変数の使用法が間違っているのでブロッキングを起こします。

例のあとに説明が続いています。その説明のあとに，デバッガを使用してスレッドの相対的な実行を制御することによってブロッキングを診断する方法を示しています。

Example 16–1 では，初期スレッドにより，計算作業を行う 2 つのワーカ・スレッドが作成されます。これらのワーカ・スレッドの作成後に SHOW TASK/ALL コマンドを実行すれば，それぞれが 1 つのスレッドに対応する 4 つのタスクが表示されます。第 16.4 節に SHOW TASK コマンドの使用法が説明されています。

- %TASK 1 が初期スレッドであり， main( )から実行される。第 16.3.3 項では，%TASK 1 などのタスク ID が定義されている。
- %TASK 2 と%TASK 3 は，ワーカ・スレッドである。

Example 16–1 では，ワーカ・スレッドのパスの行 3893 に同期化点 (条件待ち) が設けられています。行 3877 から始まるコメントは，このような直接的な呼び出しは間違ったプログラミング方法であることを示したうえで，正しいコーディング方法を示しています。

このプログラムを実行すると，ワーカ・スレッドが大量の計算を行っているときに初期スレッドが条件変数をブロードキャストします。条件変数をモニタしている最初のスレッドは初期スレッドのブロードキャストを検出してクリアし，残りのスレッドを放置します。実行が妨げられ，プログラムは終了できなくなります。

Example 16–1 C のマルチスレッド・プログラムの例

(次ページに続く)

Example 16–1 (続き) C のマルチスレッド・プログラムの例

```
3777  /* 定義  */
3778  #define NUM_WORKERS 2           /* ワーカ・スレッドの数    */
3779
3780  /* マクロ                                                 */
3781  #define check(status,string) \
3782      if (status == -1) perror (string); \
3783
3784  /* グローバル変数                                         */
3785  int              cv_pred1;     /* 条件変数の述語          */
3786  pthread_mutex_t  cv_mutex;     /* 条件変数のミューテクス  */
3787  pthread_cond_t   cv;           /* 条件変数                */
3788  pthread_mutex_t  print_mutex;  /* プリント・ミューテクス  */


3789
3790  /* ルーチン                                               */
3791  static pthread_startroutine_t
3792  worker_routine (pthread_addr_t  arg);
3793
3794  main ()
3795     {
3796     pthread_t  threads[NUM_WORKERS];  /* ワーカ・スレッド  */
3797     int        status;                /* 戻り状態値        */
3798     int        exit;                  /* Join終了状態値    */
3799     int        result;                /* Join結果値        */
3800     int        i;                     /* ループ索引        */
3801
3802     /* ミューテクスの初期化                                */
3803     status = pthread_mutex_init (&cv_mutex, pthread_mutexattr_default);
3804     check (status, "cv_mutex initilization bad status");
3805     status = pthread_mutex_init (&print_mutex, pthread_mutexattr_default);
3806     check (status, "print_mutex intialization bad status");
3807
3808     /* 条件変数の初期化                                    */
3809     status = pthread_cond_init (&cv, pthread_condattr_default);
3810     check (status, "cv condition init bad status");
3811
3812     /* 条件変数の述語の初期化                              */
3813     cv_pred1 = 1;                                          1
3814
3815     /* ワーカ・スレッドの作成                              */
3816     for (i = 0; i < num_workers; i++) {                    2
3817         status = pthread_create (
3818                         &threads[i],
3819                         pthread_attr_default,
3820                         worker_routine,
3821                         0);
3822         check (status, "threads create bad status");
3823         }
3824
```

(次ページに続く)

Example 16–1 (続き) C のマルチスレッド・プログラムの例

```
3825     /* cv_pred1を偽に設定。可視性を保つためにロック内で行う。*/
3826
3827     status = pthread_mutex_lock (&cv_mutex);
3828     check (status, "cv_mutex lock bad status");
3829
3830     cv_pred1 = 0;                                          3
3831
3832     status = pthread_mutex_unlock (&cv_mutex);
3833     check (status, "cv_mutex unlock bad status");
3834
3835     /* ブロードキャストの実施 */
3836     status = pthread_cond_broadcast (&cv);                 4
3837     check (status, "cv broadcast bad status");
3838
3839     /* 両方のワーカ・スレッドの結合を試行                    */
3840     for (i = 0; i < num_workers; i++) {                    5
3841         exit = pthread_join (threads[i], (pthread_addr_t*)&result);
3842         check (exit, "threads join bad status");
3843         }
3844     }
3845
3846  static pthread_startroutine_t
3847  worker_routine(arg)
3848     pthread_addr_t   arg;                                  6
3849     {
3850     int   sum;
3851     int   iterations;
3852     int   count;
3853     int   status;
3854
3855     /* 大量の計算を実施                                    */
3856     for (iterations = 1; iterations < 10001; iterations++) {
3857         sum = 1;
3858         for (count = 1; count < 10001; count++) {
3859             sum = sum + count;
3860             }
3861         }
3862
3863     /* Printfはリエントラントとは限らないので，一度に1スレッドを実行 */
3864
3865     status = pthread_mutex_lock (&print_mutex);
3866     check (status, "print_mutex lock bad status");
3867     printf (" The sum is %d \n", sum);
3868     status = pthread_mutex_unlock (&print_mutex);
3869     check (status, "print_mutex unlock bad status");
3870
3871     /* この条件変数のミューテクスをロックする。スレッドにより条件変数がブ*/
3872     /* ロックされるとpthread_condによりそのミューテクスがアンロックされる*/
3873
3874     status = pthread_mutex_lock (&cv_mutex);
3875     check (status, "cv_mutex lock bad status");
```

(次ページに続く)

Example 16–1 (続き) C のマルチスレッド・プログラムの例

```
3876
3877     /* 次の文では，条件待ち呼び出しのまわりをループし，その条件変数の述 */
3878     /* 語をチェックするのが正しい条件待ちの構文ということになります。    */
3879     /* そうすれば，ブロードキャスト済みの可能性がある条件変数を待った    */
3880     /* り，間違ったウェイクアップによって起動されるのを回避できます。そ */
3881     /* のスレッド がウェイクアップされ，しかもその述語が偽であれば，実  */
3882     /* 行が再開されます。正しい呼び出しは，たとえば次のようになります。  */
3883     /*                                                                    */
3884     /*    while (cv_pred1) {                                               */
3885     /*      status = pthread_cond_wait (&cv, &cv_mutex);                  */
3886     /*      check (status, "cv condition wait bad status");               */
3887     /*    }                                                               */
3888     /*                                                                    */
3889     /* 次のコーディングで使用されているような直接的な呼び出しでは，      */
3890     /* スレッドが間違ってウェイクアップされたり，この例のワーカ・        */
3891     /* スレッドの1つと同様に永続的にブロックされることがあります。        */
3892
3893     status = pthread_cond_wait (&cv, &cv_mutex);              7
3894     check (status, "cv condition wait bad status");
3895
3896     /* 条件待ちでブロックされている間，そのルーチンはミューテクスを手放 */
3897     /* しますが，制御が戻ったらミューテクスを取り出します。              */
3898
3899     status = pthread_mutex_unlock (&cv_mutex);
3900     check (status, "cv_mutex unlock bad status");
3901
3902     return (int)arg;
3903     }
```

次の番号は， Example 16–1 の番号に対応しています。

1 main( )の最初のいくつかの文では，スレッドが使用する同期化オブジェクトと条件変数に対応する述語が初期化される。それらの同期化オブジェクトは省略時の属性により初期化される。条件変数の述語は，述語のまわりをループしているスレッドがループし続けるように初期化される。プログラムのこの箇所で SHOW TASK/ALL を実行すれば，%TASK 1 が表示される。

2 ワーカ・スレッド%TASK 2 と%TASK 3 が作成される。ここで作成された各スレッドは同じ起動ルーチン (worker_routine) を実行するので， pthread_create に対する同じ呼び出しを再使用できる。ただし，異なるスレッド ID を格納するためにわずかな違いがある。それらのスレッドは省略時の属性を使用して作成され，この例では使用されない引数を引き渡される。

3 条件変数に対応する述語がブロードキャストの準備のためクリアされる。この結果，条件変数によってウェイクアップされるスレッドは正しくウェイクアップされ，間違ってウェイクアップされることはない。述語をクリアすると，条件変数がブロードキャスト済みまたはシグナル通知済みとなっているので，新しいスレッドが条件変数を待つこともなくなる。期待通りの効果が得られるかどうかは，

行 3893 の条件待ち呼び出しのコーディングが正しいかどうかによるが，この例のコーディングは間違っている。

4 初期スレッドはほとんどすぐにブロードキャスト呼び出しを実行するので，どのワーカ・スレッドもまだ条件待ちをしていない。ブロードキャストにより，その時点でその条件変数を待っているすべてのスレッドがウェイクアップされる。

プログラマは，ブロードキャスト時にすべてのスレッドが条件変数を待っているようにするか，またはブロードキャストがすでに起こったことを対応する述語で明らかにすることによって，そのブロードキャストが確実に認識されるようにしなければならない。このような方法は，この例では意図的に省いている。

5 初期スレッドは，ワーカ・スレッドがどちらも正しく終了したことを確かめるために，両者を結合しようとする。

6 ワーカ・スレッドが worker_routine を実行すると，大量の計算に時間がかかる。そのため初期スレッドは，どちらのワーカ・スレッドも条件変数を待つ準備ができていないときにその条件変数をブロードキャストする。

7 次にワーカ・スレッドは pthread_cond_wait 呼び出しを実行し，必要に応じて呼び出しのまわりでロックを行う。両方のワーカ・スレッドがブロードキャストを検出できなくてブロックするのはこの箇所である。そのときに SHOW TASK/ALL コマンドを入力すれば，両方のワーカ・スレッドが条件変数を待っていることが分かる。このようにプログラムがデッドロック状態になったときに制御をデバッガに戻すには， Ctrl/C を押さなければならない。

デバッガを使用すればスレッドの相対的な実行を制御することにより，Example 16–1 のような問題を診断することができます。この例の場合は，初期スレッドの実行を中断してワーカ・スレッドに計算を終了させ，ワーカ・スレッドがブロードキャスト時に条件変数を待っているようにできます。その手順は次のとおりです。

1. デバッグ・セッションの開始時に，ブロードキャストの直前で初期スレッドの実行が中断するよう，行 3836 にブレークポイントを設定する。
2. 初期スレッドを実行しワーカ・スレッドを作成する GO コマンドを入力する。
3. すべてのスレッドの実行を中断するこのブレークポイントで， SET TASK/HOLD %TASK 1 コマンドによって初期スレッドを保留する。
4. ワーカ・スレッドが実行を続けるように GO コマンドを入力する。初期スレッドは保留され，実行できない。
5. ワーカ・スレッドが条件変数をブロックしているときは，その時点で Ctrl/C を押せば制御はデバッガに戻る。 SHOW TASK/ALL コマンドを実行すれば，両方のワーカ・スレッドが条件待ち副次状態で中断していることが示される。示されない場合は，それらのワーカ・スレッドを実行する GO コマンドを入力し， Ctrl/C を押してから SHOW TASK/ALL を入力する。両方のワーカ・スレッドが条件待ち副次状態になるまでこの手順を繰り返す。

6. 最初に SET TASK/NOHOLD %TASK 1 コマンド，次に初期スレッドが実行を再開してブロードキャストを行うように， GO コマンドを入力する。これで，ワーカ・スレッドは結合し正常終了する。

16.2.2 Ada のタスキング・プログラムの例

Example 16–2 はデバッグ中のタスキング・プログラムによくあるエラーを示します。ここで示すのは， OpenVMS Alpha 上で動作している OpenVMS デバッガの例です。この例のプロシージャ BREAK の呼び出しは，ブレークポイントを設定したり各タスクの状態を観察する候補箇所です。この例をデバッガ制御の下で実行する場合は，プロシージャ BREAK の各呼び出しの箇所で次のコマンドを入力してブレークポイントを設定し，それぞれのブレークポイントで各タスクのそのときの状態を表示できます。

```
DBG> SET BREAK %LINE  37 DO (SHOW TASK/ALL)
DBG> SET BREAK %LINE  61 DO (SHOW TASK/ALL)
DBG> SET BREAK %LINE  65 DO (SHOW TASK/ALL)
DBG> SET BREAK %LINE  81 DO (SHOW TASK/ALL)
DBG> SET BREAK %LINE  87 DO (SHOW TASK/ALL)
DBG> SET BREAK %LINE  91 DO (SHOW TASK/ALL)
DBG> SET BREAK %LINE 105 DO (SHOW TASK/ALL)
```

このプログラムでは次の 4 つのタスクが作成されます。

- メイン・プログラム TASK_EXAMPLE を実行する環境タスク。このタスクが最初に作成され，そのあとでライブラリ・パッケージ (この例では， TEXT_IO) が詳細化される。 SHOW TASK を実行すれば，環境タスクのタスク ID は%TASK 1 と表示される。
- FATHER というタスク・オブジェクト。このタスクはメイン・プログラムによって宣言され， SHOW TASK を実行すれば%TASK 3 と表示される。
- CHILD というタスク。このタスクはタスク FATHER によって宣言され， SHOW TASK を実行すれば%TASK 4 と表示される。
- MOTHER というタスク。このタスクはメイン・プログラムによって宣言され，SHOW TASK を実行すれば%TASK 2 と表示される。

Example 16–2 Ada のタスキング・プログラムの例

(次ページに続く)

Example 16–2 (続き) Ada のタスキング・プログラムの例

```
1 --Tasking program that demonstrates various tasking conditions.
2
3 with TEXT_IO; use TEXT_IO;
4 procedure TASK_EXAMPLE is 1
5
6   pragma TIME_SLICE(0.0); -- Disable time slicing. 2
7
8    task type FATHER_TYPE is
9       entry START;
10      entry RENDEZVOUS;
11      entry BOGUS; -- Never accepted, caller deadlocks.
12   end FATHER_TYPE;
13
14   FATHER : FATHER_TYPE; 3
15
16   task body FATHER_TYPE is
17      SOME_ERROR : exception;
18
19      task CHILD is 4
20         entry E;
21      end CHILD;
22
23      task body CHILD is
24      begin
25            FATHER_TYPE.BOGUS; -- Deadlocks on call to its parent.
26      end CHILD;                  -- Whenever a task-type name
27                                        -- (here, FATHER_TYPE) is used within the
28                                        -- task body, the name denotes the task
29                                        -- currently executing the body.
```

(次ページに続く)

Example 16–2 (続き) Ada のタスキング・プログラムの例

```
30   begin -- (of FATHER_TYPE body)
31
32      accept START do
33         -- Main program is now waiting for this rendezvous completion,
34                    -- and CHILD is suspended  when it calls the entry BOGUS.
35
36      null;
37 <<B1>>    end START;
38
39     PUT_LINE("FATHER is now active and"); 5
40     PUT_LINE("is going to rendezvous with main program.");
41
42     for I in 1..2 loop
43        select
44           accept RENDEZVOUS do
45              PUT_LINE("FATHER now in rendezvous with main program");
46           end RENDEZVOUS;
47       or
48           terminate;
49        end select;
50
51        if I = 2 then
52           raise SOME_ERROR;
53        end if;
54    end loop;
55
56  exception
57    when OTHERS =>
58                -- CHILD is suspended on entry call to BOGUS.
59                -- Main program is going to delay while FATHER terminates.
60                -- Mother in suspended state with "Not yet activated" sub state.
61<<B2>>         abort CHILD;
62                -- CHILD is now abnormal due to the abort statement.
63
64
65<<B3>>        raise; -- SOME_ERROR exception terminates
66               FATHER.
67  end FATHER_TYPE; 6
68
69  task MOTHER is 7
70     entry START;
71    pragma PRIORITY (6);
72  end MOTHER;
```

(次ページに続く)

Example 16–2 (続き) Ada のタスキング・プログラムの例

```
73
74  task body MOTHER is
75  begin
76     accept START;
77           -- At this point, the main program is waiting for its dependents
78               -- (FATHER and MOTHER) to terminate.   FATHER is terminated.
79
80    null;
81<<B4>>  end MOTHER;
82
83 begin    -- (of TASK_EXAMPLE)8
84          -- FATHER is suspended at accept start, and
85          -- CHILD is suspended in its deadlock.
86          -- Mother in suspended state with "Not yet activated" sub state.
87<<B5>>    FATHER.START; 9
88           -- FATHER is suspended at the 'select' or 'terminate' statement.
89
90
91<<B6>>    FATHER.RENDEZVOUS;
92    FATHER.RENDEZVOUS; 10
93    loop 11
94          -- This loop causes the main program to busy wait for termination of
95          -- FATHER, so that FATHER can be observed in its terminated state.
96       if FATHER'TERMINATED then
97          exit;
98       end if;
99       delay 10.0;       -- 10.0 so that MOTHER is suspended
100   end loop;            -- at the 'accept' statement (increases determinism).
101
102   -- FATHER has terminated by now with an unhandled
103                -- exception, and CHILD no longer exists because its
104                -- master (FATHER) has terminated.  Task MOTHER is ready.
105<<B7>>   MOTHER.START; 12
106             -- The main program enters a wait-for-dependents state
107             -- so that MOTHER can finish executing.
108 end TASK_EXAMPLE; 13
1 --Tasking program that demonstrates various tasking conditions.
2
3 with TEXT_IO; use TEXT_IO;
4 procedure TASK_EXAMPLE is 1
5
6   pragma TIME_SLICE(0.0); -- Disable time slicing. 2
7
8    task type FATHER_TYPE is
9       entry START;
10      entry RENDEZVOUS;
11      entry BOGUS; -- Never accepted, caller deadlocks.
12   end FATHER_TYPE;
```

(次ページに続く)

Example 16–2 (続き) Ada のタスキング・プログラムの例

```
13
14   FATHER : FATHER_TYPE; 3
15
16   task body FATHER_TYPE is
17      SOME_ERROR : exception;
18
19      task CHILD is 4
20         entry E;
21      end CHILD;
22
23      task body CHILD is
24      begin
25            FATHER_TYPE.BOGUS; -- Deadlocks on call to its parent.
26      end CHILD;                   -- Whenever a task-type name
27                                         -- (here, FATHER_TYPE) is used within the
28                                         -- task body, the name denotes the task
29                                         -- currently executing the body.
30   begin -- (of FATHER_TYPE body)
31
32      accept START do
33         -- Main program is now waiting for this rendezvous completion,
34                   -- and CHILD is suspended  when it calls the entry BOGUS.
35
36      null;
37 <<B1>>    end START;
38
39     PUT_LINE("FATHER is now active and"); 5
40     PUT_LINE("is going to rendezvous with main program.");
41
42     for I in 1..2 loop
43        select
44           accept RENDEZVOUS do
45              PUT_LINE("FATHER now in rendezvous with main program");
46           end RENDEZVOUS;
47       or
48           terminate;
49        end select;
50
51        if I = 2 then
52           raise SOME_ERROR;
53        end if;
54    end loop;
55
56  exception
57    when OTHERS =>
58                -- CHILD is suspended on entry call to BOGUS.
59                -- Main program is going to delay while FATHER terminates.
60                -- Mother in suspended state with "Not yet activated" sub state.
61<<B2>>         abort CHILD;
62                -- CHILD is now abnormal due to the abort statement.
63
```

(次ページに続く)

Example 16–2 (続き) Ada のタスキング・プログラムの例

```
64
65<<B3>>       raise; -- SOME_ERROR exception terminates
66              FATHER.
67  end FATHER_TYPE; 6
68
69  task MOTHER is 7
70     entry START;
71    pragma PRIORITY (6);
72  end MOTHER;
73
74  task body MOTHER is
75  begin
76     accept START;
77           -- At this point, the main program is waiting for its dependents
78               -- (FATHER and MOTHER) to terminate.   FATHER is terminated.
79
80    null;
81<<B4>>  end MOTHER;
82
83 begin    -- (of TASK_EXAMPLE)8
84          -- FATHER is suspended at accept start, and
85          -- CHILD is suspended in its deadlock.
86          -- Mother in suspended state with "Not yet activated" sub state.
87<<B5>>    FATHER.START; 9
88           -- FATHER is suspended at the 'select' or 'terminate' statement.
89
90
91<<B6>>    FATHER.RENDEZVOUS;
92    FATHER.RENDEZVOUS; 10
93    loop 11
94          -- This loop causes the main program to busy wait for termination of
95          -- FATHER, so that FATHER can be observed in its terminated state.
96       if FATHER'TERMINATED then
97          exit;
98       end if;
99       delay 10.0;       -- 10.0 so that MOTHER is suspended
100   end loop;            -- at the 'accept' statement (increases determinism).
101
102   -- FATHER has terminated by now with an unhandled
103                -- exception, and CHILD no longer exists because its
104                -- master (FATHER) has terminated.  Task MOTHER is ready.
105<<B7>>   MOTHER.START; 12
106             -- The main program enters a wait-for-dependents state
107             -- so that MOTHER can finish executing.
108 end TASK_EXAMPLE; 13
```

次の番号は， Example 16–2 の番号に対応しています。

1 すべての Ada ライブラリ・パッケージ (この場合は TEXT_IO) の作成が終わると，メイン・プログラムが自動的に呼び出されてその宣言部分の作成が開始される (行 5 〜 82)。

2 この例では，実行のつど同じ処理が行われるように，タイム・スライス機能 (第 16.5.2 項を参照) は使用していない。プラグマ TIME_SLICE の値が 0.0 になっているのは，プロシージャ TASK_EXAMPLE のためにタイム・スライス機能を禁止する必要があることを示している。

VAX プロセッサでは，プラグマ TIME_SLICE を省略するか値 0.0 を指定すると，タイム・スライス機能は禁止される。

Alpha プロセッサでは，タイム・スライス機能を禁止するためにプラグマ TIME_SLICE(0.0) を使用しなければならない。

3 タスク・オブジェクト FATHER が作成され，%TASK 2 と指定されたタスクが作成される。 FATHER にはプラグマ PRIORITY がないので，省略時の優先順位が与えられる。 FATHER (%TASK 2) は中断状態で作成され， Ada の規則に従ってメイン・プログラムの文部分が開始されて初めて起動されます (行 83)。行 29 ～ 81 のタスク本体の作成では， FATHER_TYPE 型のタスクが実行する文が定義されている。

4 タスク FATHER はタスク CHILD を宣言する (行 32)。 1 つのタスクは， 1 つのタスク・オブジェクトであり，なにか 1 つのタスク型を表す。タスク CHILD が作成され，起動されるのは， FATHER が起動されてからである。

5 非同期システム・トラップ (AST) を引き起こすのは，この TEXT_IO の一連の PUT_LINE 文だけである。入出力 (I/O) の終了により AST が実行要求される。

6 タスクの FATHER は，メイン・プログラムが待っているときに並行して起動される。 FATHER にはプラグマ PRIORITY がないので，省略時の優先順位 7 を与えられる (省略時の優先順位については『DEC Ada Language Reference Manual』を参照)。 FATHER の起動は行 29 ～ 44 で作成される。

起動されたタスク FATHER は，タスク CHILD が起動され，%TASK 3 を指定されたタスクが作成されるのを待つ。 CHILD は行 38 で 1 つのエントリ呼び出しを実行し，そのエントリが受け付けられないのでデッドロックになる (第 16.7.1 項を参照)。

タイム・スライス機能が禁止され，優先順位の高い実行可能なタスクがないので， FATHER は起動後も行 47 の ACCEPT 文でブロックされるまで実行される。

7 タスク MOTHER が定義され，%TASK 4 と指定されたタスクが作成される。プラグマ PRIORITY により， MOTHER には優先順位 6 が与えられる。

8 タスク MOTHER が起動し，行 91 を実行する。起動後，メイン・プログラム (%TASK 1) の実行再開が可能になる。%TASK 1 は省略時の優先順位が 7 であり， MOTHER の優先順位より高いので，メイン・プログラムの実行が再開される。

9 メイン・プログラムとタスク FATHER の最初のランデブ。この後， FATHER は行 58 の TARMINATO 文の SELECT で中断される。

10 FATHER との 3 回目のランデブでは， FATHER は行 67 で SOME_ERROR という例外を発生させる。ハンドラは行 72 でその例外を捉え，中断しているタスク CHILD を強制終了してから，再び例外を発生させる。その後， FATHER は終了する。

11 delay 文で指定されたループにより，制御が行 122 に到達するときには FATHER は終了するのに十分なほど先まで実行されている。

12 このエントリ呼び出しにより， MOTHER は行 93 のランデブ待ちを解除される。MOTHER はその accept 文 (その他の文は含まない) を実行し，ランデブは終了する。すると，優先順位が 6 にすぎない MOTHER は行 94 で制御を奪われる。

13 メイン・プログラム (%TASK 1) は MOTHER とのランデブ後，行 127 ～ 129 を実行する。メイン・プログラムは行 129 で，自分のすべての依存タスクの終了を待たなければならない (第 16.6.4 項を参照)。メイン・プログラムが行 129 に到達するとき，まだ終了していないタスクは MOTHER だけである。 MOTHER は，行 97 の**空**文が実行されるまでは終了できない。 MOTHER は行 98 で実行を終了する。すべてのタスクが終了したので，メイン・プログラムは実行を終了する。メイン・プログラムから制御が戻されて，コマンド行インタプリタの実行が再開される。

16.3 デバッガ・コマンドによるタスクの指定

タスクとは，その他のタスクと並行して実行される要素です。タスクには固有のタスク ID (第 16.3.3 項を参照)，独立したスタック，および独立したレジスタ・セットが与えられます。

アクティブ・タスクと可視タスクの現在の定義により，タスク操作のコンテキストが決まります。第 16.3.1 項を参照してください。

デバッガ・コマンドにタスクを指定するときには，次のいずれかの形式で指定できます。

- プログラム内に宣言されているタスク (スレッド) 名 (たとえば，第 16.2.2 項の FATHER) またはタスク値を算出するための言語式。第 16.3.2 項には，タスク用の Ada 言語式が説明されている。
- タスク ID (たとえば，%TASK 2)。第 16.3.3 項を参照。
- タスク組み込みシンボル (たとえば，%ACTIVE_TASK)。第 16.3.4 項を参照。

16.3.1 アクティブ・タスクと可視タスクの定義

アクティブ・タスクとは，STEP，GO，CALL，または EXIT コマンドを実行したときに起動されるタスクです。プログラムをデバッガの制御下に置くと，最初はアクティブ・タスクの中で実行が中断されます。デバッグ・セッション中にアクティブ・タスクを変更するには SET TASK/ACTIVE コマンドを使用します。

注意

SET TASK/ACTIVE コマンドは，POSIX Threads (OpenVMS VAX システム，Alpha システム，および Integrity システム) と POSIX Threads 経由で実行するタスキングである Ada (OpenVMS Alpha システムおよび Integrity システム) では動作しません。POSIX Threads で照会型のアクションを行うときは，SET TASK/ACTIVE コマンドの代わりに，SET TASK/VISIBLE コマンドを使用します。特定のスレッドでステップを制御したいときは，ブレークポイントを適切な位置に配置します。

次のコマンドでは CHILD というタスクがアクティブ・タスクになります。

```
DBG> SET TASK/ACTIVE CHILD
```

可視タスクとは，スタックとレジスタ・セットがデバッガによって現在のコンテキストとして使用されるタスクです。デバッガはシンボル，レジスタ値，ルーチン呼び出し，ブレークポイントなどを参照するときにスタックとレジスタ・セットを使用します。たとえば，次のコマンドでは，可視タスクのコンテキストの変数 KEEP_COUNT の値が表示されます。

```
DBG> EXAMINE KEEP_COUNT
```

最初は，可視タスクがアクティブ・タスクです。可視タスクを変更するには，SET TASK/VISIBLE コマンドを使用します。このコマンドにより，アクティブ・タスクに影響を与えずにその他のタスクの状態を参照することができます。

デバッガ・コマンドに組み込みシンボルの%ACTIVE_TASK と%VISIBLE_TASK を使用することにより，それぞれアクティブ・タスクと可視タスクを指定できます (第 16.3.4 項を参照)。

SET TASK コマンドによるタスク特性の変更についての詳しい説明は，第 16.5 節を参照してください。

16.3.2 Ada のタスキングの構文

タスクを宣言するには，単一タスクを宣言するか，またはあるタスク型のオブジェクトを宣言します。次に例を示します。

```
-- タスク型の宣言。
--
task type FATHER_TYPE is
 . . .
end FATHER_TYPE;

task body FATHER_TYPE is
 . . .
end FATHER_TYPE;

-- 単一タスク。
--
task MOTHER is
 . . .
end MOTHER;

task body MOTHER is
 . . .
end MOTHER;
```

タスク・オブジェクトとは，タスク値を含むデータ項目です。タスク・オブジェクトが作成されるのは，プログラムによって単一タスクかタスク・オブジェクトが作成されるとき，タスク構成要素を含んでいる配列かレコードをユーザが宣言するとき，またはタスク・アロケータが評価されるときです。次に例を示します。

```
-- タスク・オブジェクトの宣言。
--
FATHER : FATHER_TYPE;

-- タスク・オブジェクト(T)はレコードの構成要素。
--
type SOME_RECORD_TYPE is
   record
      A, B: INTEGER;
      T   : FATHER_TYPE;
   end record;

HAS_TASK : SOME_RECORD_TYPE;

-- タスク・オブジェクト(POINTER1)はアロケータを通じて作成される。
--
type A is access FATHER_TYPE;
POINTER1 : A := new FATHER_TYPE;
```

タスク・オブジェクトは，その他のオブジェクトに似ています。デバッガ・コマンドにタスク・オブジェクトを指定するときには，名前かパス名を指定します。次に例を示します。

```
DBG> EXAMINE FATHER
DBG> EXAMINE FATHER_TYPE$TASK_BODY.CHILD
```

タスク・オブジェクトを作成すると， Ada 実行時ライブラリによってタスクが作成され，そのタスク・オブジェクトにタスク値が割り当てられます。タスク・オブジェクトの値はその他の Ada オブジェクトと同じく，オブジェクトが初期化されるまでは未定義になるので，初期化されていない値を使用するとその結果は予測できません。

あるタスク型または単一タスクの**タスク本体**は，1つのプロシージャとしてAdaの中に組み込まれます。そのプロシージャはその型のタスクが起動されるとき，Adaの実行時ライブラリから呼び出されます。デバッガはタスク本体を普通のAdaプロシージャとして処理します。特別な構造の名前を持っている点が異なります。

デバッガ・コマンドにタスク本体を指定するには，次の構文を使用して，タスク型として宣言されているタスクを指定します。

task-type-identifier$TASK_BODY

単一タスクの指定には，次の構文を使用します。

task-identifier$TASK_BODY

たとえば，次のように指定します。

```
DBG> SET BREAK FATHER_TYPE$TASK_BODY
```

デバッガはタスク依存のAda属性T'CALLABLE，E'COUNT，T'STORAGE_SIZE，およびT'TERMINATEDはサポートしません。このうち，Tはタスク型，Eはタスク・エントリです(これらの属性についての詳しい説明は，Adaの資料を参照してください)。EVALUATE CHILD'CALLABLEなどのコマンドは入力できません。しかし，デバッガのSHOW TASKコマンドを使用してこれらの属性の内容を知ることはできます。詳しい説明は，第16.4節を参照してください。

16.3.3 タスクID

タスクIDとは，タスクがタスキング・システムによって作成されるときタスクに付けられる番号です。タスクIDにより，タスクはプログラムの実行中は常に一意的に識別されます。

タスクIDの構文は次のとおりです。ただし，*n*は正の10進整数です。

```
%TASK n
```

あるタスク・オブジェクトのタスクIDを知るためには，そのタスク・オブジェクトを評価または検査します。次はその一例です。パス名はAdaの構文に従っています。

```
DBG> EVALUATE FATHER
%TASK 2
DBG> EXAMINE FATHER
TASK_EXAMPLE.FATHER:  %TASK 2
```

プログラミング言語に組み込みタスキング・サービスが用意されていない場合，タスクのタスクIDを得るためにはEXAMINE/TASKコマンドを使用しなければなりません。

EXAMINE/TASK/HEXADECIMAL コマンドにタスク・オブジェクトを指定すると 16 進のタスク値が表示されるので注意してください。タスク値とはそのタスクのタスク (すなわちスレッド) 制御ブロックのアドレスです。次はその一例です (Ada の例)。

```
DBG> EXAMINE/HEXADECIMAL FATHER
TASK_EXAMPLE.FATHER: 0015AD00
DBG>
```

SHOW TASK/ALL コマンドでは，現存するすべてのタスクに割り当てられているタスク ID を表示することができます。これらのタスクの中には，次のような理由からユーザにとってはなじみのないものもあります。

- SHOW TASK/ALL の表示には，ユーザのアプリケーションに関連したタスクだけでなく， POSIX Threads，リモート・プロシージャ呼び出しサービス， C 実行時ライブラリのようなサブシステムが作成したタスクも含まれる。
- SHOW TASK/ALL の表示には，ユーザのオペレーティング・システム，ユーザのタスキング・サービス，および作成用サブシステムに依存するタスク ID 割り当てが含まれる。異なったシステムで実行される同一のタスキング・プログラムや異なったサービス用に調整された同一のタスキング・プログラムは，別のタスクを同じ 10 進整数では識別しない。%TASK1 だけは例外であり，このタスクはすべてのシステムおよびサービスによって，メイン・プログラムを実行するタスクに割り当てられる。

次の各例は，それぞれ Example 16–1 と Example 16–2 で実行したときのものです。

```
DBG> SHOW TASK/ALL
  task id    state hold  pri substate        thread_object
  %TASK    1 READY HOLD   12                 Initial thread
  %TASK    2 SUSP         12 Condition Wait  THREAD_EX1\main\threads[0].field1
  %TASK    3 SUSP         12 Condition Wait  THREAD_EX1\main\threads[1].field1
DBG>

DBG> SHOW TASK/ALL
  task id pri hold state   substate     task object
* %TASK 1  7       RUN                SHARE$ADARTL+130428
  %TASK 2  7       SUSP  Accept       TASK_EXAMPLE.MOTHER+4
  %TASK 4  7       SUSP  Entry call   TASK_EXAMPLE.FATHER_TYPE$TASK_BODY.CHILD+4
  %TASK 3  6       READY              TASK_EXAMPLE.MOTHER+4
DBG>
```

タスク ID を使用すれば，デバッガの条件文に非存在タスクを指定できます。たとえば，自分のプログラムを一度実行して，%TASK 2 と%TASK 3 を調べたい場合は，その次のデバッグ・セッションを開始して，まだ%TASK 2 も%TASK 3 も作成されていないときに次のコマンドを入力します。

```
DBG> SET BREAK %LINE 60 WHEN (%ACTIVE_TASK=%TASK 2)
DBG> IF (%CALLER=%TASK 3) THEN (SHOW TASK/FULL)
```

タスクが作成される前にそのタスク ID を特定のデバッガ・コマンドに指定しても，デバッガがエラーを報告することはありません。しかし，タスク・オブジェクトが存在する前にそのタスク・オブジェクト名を使用すると，デバッガによりエラーが報告されます。タスクは作成されて初めて存在することになります。タスクは終了後しばらくして非存在になります。非存在タスクがデバッガの SHOW TASK コマンドによって表示されることはありません。

プログラムの文が同じ順序で実行されるかぎり，そのプログラムを実行するつど同じタスクには同じタスク ID が割り当てられます。しかし， AST (delay 文の満了や入出力 (I/O) の完了で起こる) が異なる順序で発生するために実行順序が変わることがあります。タイム・スライス機能を許可しても実行順序が変わることがあります。同じプログラムの実行中に同じタスク ID が 2 回以上割り当てられることはありません。

16.3.4 タスク組み込みシンボル

表 16–2 に定義されているデバッガの組み込みシンボルを使用すれば，コマンド・プロシージャやコマンド構造にタスクを指定できます。

表 16–2 タスク組み込みシンボル

組み込みシンボル	機能
%ACTIVE_TASK	GO, STEP, CALL，または EXIT コマンドの実行によって起動されるタスク。
%CALLER_TASK	(Ada プログラムだけの機能。) 実行される accept 文のエントリを呼び出したタスク。
%NEXT_TASK	デバッガのタスク・リスト内の，可視タスクの後のタスク。各タスクの順序はランダムだが，同じプログラムを 1 回実行している間は首尾一貫している。
%PREVIOUS_TASK	デバッガのタスク・リスト内の，可視タスクの前のタスク。
%VISIBLE_TASK	シンボル，レジスタ値，ルーチン呼び出し，ブレークポイントなどの参照に現在のコンテキストとして使用される呼び出しスタックとレジスタ・セットを持っているタスク。

これらのタスク組み込みシンボルの使用例は次のとおりです。

次のコマンドでは，可視タスクのタスク ID が表示されます。

```
DBG> EVALUATE %VISIBLE_TASK
```

次のコマンドではアクティブ・タスクが保留されます。

```
DBG> SET TASK/HOLD %ACTIVE_TASK
```

次のコマンドでは行 38 にブレークポイントが設定されます。このブレークポイントはタスク CHILD が行 38 を実行するときにだけ検出されます。

```
DBG> SET BREAK %LINE 38 WHEN (%ACTIVE_TASK=CHILD)
```

シンボル%NEXT_TASK と%PREVIOUS_TASK を使用すれば，現存しているすべてのタスクを順次表示できます。たとえば，次のように使用します。

```
DBG> SHOW TASK %VISIBLE_TASK; SET TASK/VISIBLE %NEXT_TASK
DBG> SHOW TASK %VISIBLE_TASK; SET TASK/VISIBLE %NEXT_TASK
   .
   .
   .
DBG> EXAMINE MONITOR_TASK
MOD\MONITOR_TASK: %TASK 2
DBG> WHILE %NEXT_TASK NEQ %ACTIVE DO (SET TASK %NEXT_TASK; SHOW CALLS)
```

16.3.4.1 呼び出し元のタスクのシンボル (Ada 専用)

シンボル%CALLER_TASK は Ada のタスクだけに使用します。これが評価されると， accept 文のエントリを呼び出したタスクのタスク ID か，%TASK 0 になります。たとえば，アクティブ・タスクが accept 文に対応する一連の文を実行していないときに%CALLER_TASK が評価されると，%TASK 0 になります。

たとえば， Example 16–2 の行 61 (accept 文の中) にブレークポイントが設定されているものとします。この例では，メイン・プログラム (%TASK 1) がエントリ RENDEZVOUS を呼び出すのに応じて，タスク FATHER(%TASK 2) が accept 文を実行します。したがって，そのときに EVALUATE %CALLER_TASK コマンドを入力すれば，呼び出しタスク，つまりメイン・プログラムのタスク ID と評価されます。

```
DBG> EVALUATE %CALLER_TASK
%TASK 1
DBG>
```

AST によるエントリ呼び出しの結果がランデブのときに%CALLER_TASK を評価すると，%TASK 0 になります。呼び出し元がタスクでないからです。

16.4 タスク情報の表示

プログラム内のタスクの情報を表示するには， SHOW TASK コマンドを使用します。

SHOW TASK コマンドでは，存在している (終了していない) タスクの情報が表示されます。省略時の設定では，可視タスクの情報が 1 行だけ表示されます。

第 16.4.1 項と第 16.4.2 項には，それぞれ， SHOW TASK コマンドによって表示される POSIX Threads タスクと Ada タスクの情報が説明されています。

16.4.1 POSIX Threads タスクのタスク情報の表示

SHOW TASK コマンドでは，プログラム内に現存しているすべてのタスクの情報が表示されます (Example 16–3 を参照)。

Example 16–3 POSIX Threads タスクに対して SHOW TASK/ALL を実行したときの表示例

```
 1          2      3      4    5                6
 task id    state hold  pri substate        thread_object
  %TASK     1 SUSP         12 Condition Wait  Initial thread
  %TASK     2 SUSP         12 Mutex Wait      T_EXAMP\main\threads[0].field1
  %TASK     3 SUSP         12 Delay           T_EXAMP\main\threads[1].field1
  %TASK     4 SUSP         12 Mutex Wait      T_EXAMP\main\threads[2].field1
* %TASK     5 RUN          12                 T_EXAMP\main\threads[3].field1
  %TASK     6 READY        12                 T_EXAMP\main\threads[4].field1
  %TASK     7 SUSP         12 Mutex Wait      T_EXAMP\main\threads[5].field1
  %TASK     8 READY        12                 T_EXAMP\main\threads[6].field1
  %TASK     9 TERM         12 Term. by alert  T_EXAMP\main\threads[7].field1
DBG>
```

次の番号は， Example 16–3 の番号に対応しています。

1 タスク ID (第 16.3.3 項を参照)。アクティブ・タスクには左端の欄にアスタリスク(*)が付けられる。

2 タスクの現在の状態 (表 16–3 を参照)。 RUN (RUNNING) 状態のタスクがアクティブ・タスクである。表 16–3 には，プログラムの実行中にどの状態に移ることができるかが示されている。

3 タスクが SET TASK/HOLD コマンドによって保留されているかどうか。タスクの保留については第 16.5.1 項を参照。

4 タスクの優先順位。

5 タスクの現在の副次状態。副次状態はあるタスク状態を引き起こした原因を推定するのに役立つ。表 16–4 を参照。

6 タスク (スレッド) オブジェクトのデバッガのパス名，またはデバッガがタスク・オブジェクトをシンボル化できない場合はタスク・オブジェクトのアドレス。

表 16–3 一般的なタスクの状態

タスクの状態	説明
RUNNING	タスクは現在プロセッサで実行中である。これがアクティブ・タスクである。この状態のタスクが移ることができるのは，READY, SUSPENDED, TERMINATED のいずれかの状態である。
READY	タスクは実行適格であり，プロセッサが使用可能になるのを待っている。この状態のタスクが移ることができるのは RUNNING 状態だけである。
SUSPENDED	タスクは中断している。つまり，プロセッサが使用できるのを待っているのではなく，あるイベントを待っている。たとえば，タスクは作成されてから起動されるまで中断状態に留まる。この状態のタスクが移ることができるのは，READY 状態または TERMINATED 状態だけである。
TERMINATED	タスクは終了している。この状態のタスクは，ほかの状態に移ることはできない。

表 16–4 POSIX Threads タスクの副次状態

タスクの副次状態	説明
Condition Wait	タスクは POSIX Threads の条件変数を待っている。
Delay	タスクは POSIX Threads 遅延への呼び出しで待っている。
Mutex Wait	タスクは POSIX Threads ミューテクスを待っている。
Not yet started	タスクはまだその起動ルーチンを実行していない。
Term. by alert	タスクは警告処理により打ち切られた。
Term. by exc	タスクは例外により打ち切られた。
Timed Cond Wait	タスクは時限 POSIX Threads 条件変数を待っている。

SHOW TASK/FULL コマンドでは，選択した各タスクについての詳細情報が表示されます。Example 16–4 に，POSIX Threads タスクの例でこのコマンドを実行した場合の出力を示します。

Example 16–4 POSIX Threads タスクに対して SHOW TASK/FULL を実行したときの表示例

```
1  task id    state hold  pri substate        thread_object
 %TASK    4 SUSP         12 Delay           T_EXAMP\main\threads[1].field1
2          Alert is pending
           Alerts are deferred

3          Next pc:           SHARE$CMA$RTL+46136
           Start routine:     T_EXAMP\thread_action
4          Scheduling policy: throughput

5          Stack storage:
           Bytes in use:          1288   6  Base:    00334C00
           Bytes available:      40185        SP:      003346F8
           Reserved Bytes:       10752        Top:     00329A00
           Guard Bytes:           4095
```

(次ページに続く)

Example 16–4（続き） POSIX Threads タスクに対して SHOW TASK/FULL を実行したときの表示例

```
7         Thread control block:
          Size:                  293        Address: 00311B78

8     Total storage:          56613
DBG>
```

次の番号は， Example 16–4 の番号に対応しています。

1 タスク情報の種別を示す。

2 なんらかの異常についての掲示板型の情報。

3 次に実行される箇所を示す PC 値と起動ルーチン。

4 タスク・スケジューリングの方針。

5 スタック記憶域の情報。

- "Bytes in use:"スタックに現在割り当てられているバイト数。
- "Bytes available:"未使用空間のバイト数。
- "Reserved bytes:"スタック・オーバフロー処理のために割り当てられている記憶域。
- "Guard bytes: "スタックの保護領域つまり書き込み禁止領域のサイズ。

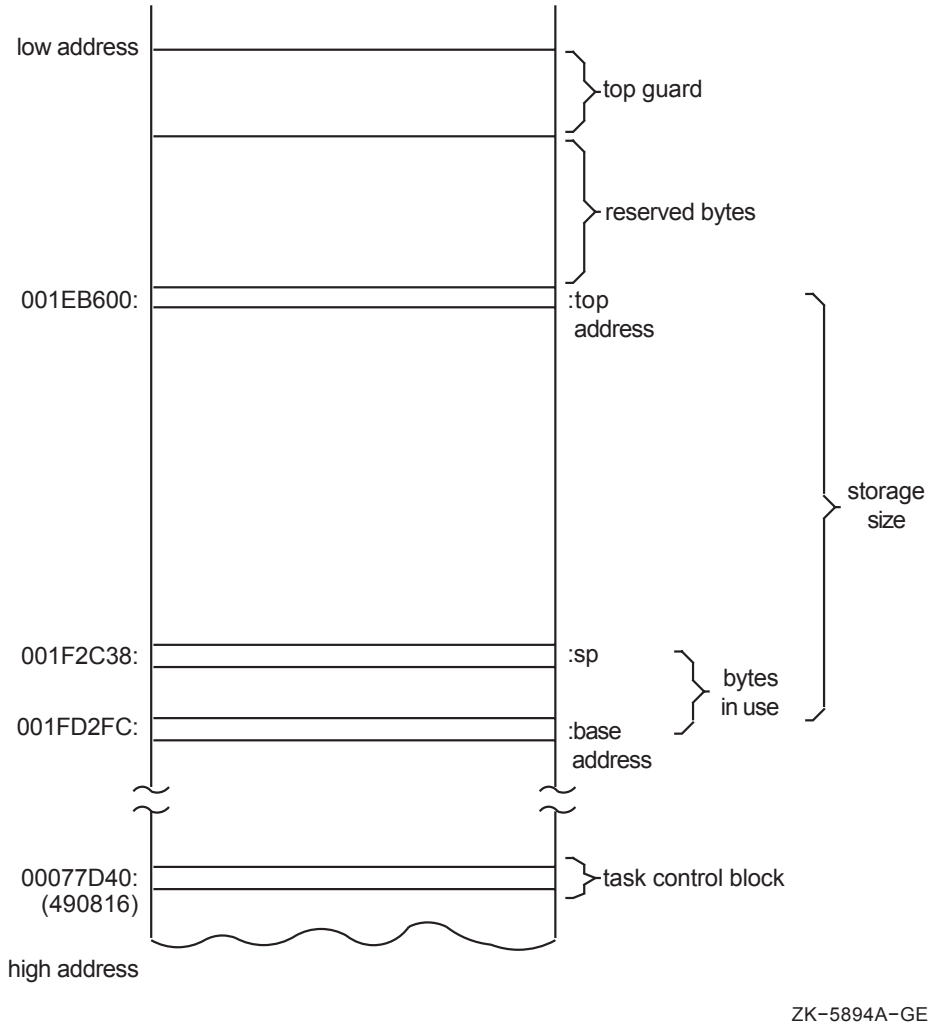
6 タスク・スタックの上限アドレスと下限アドレス。

7 タスク (スレッド) 制御ブロック情報。タスク値はタスク制御ブロックの 16 進表記アドレス。

8 タスクが使用する記憶域の合計。タスク制御ブロック・サイズ，予約バイト数，上部保護領域サイズ，記憶域サイズを合計したもの。

図 16–1 にタスク・スタックを示します。

図 16–1 タスク・スタック図



SHOW TASK/STATISTICS コマンドでは，プログラム内のすべてのタスクについての統計情報が報告されます。 Example 16–5 には， POSIX Threads タスクのプログラム例に対して SHOW TASK/STATISTICS/FULL コマンドを実行したときの出力が示されています。この情報により，プログラムの性能を測定できます。スケジューリング (コンテキスト・スイッチとも呼ぶ) の合計回数が多いほど，タスキング・オーバヘッドは大きくなります。

Example 16–5 POSIX Threads タスクに対して SHOW TASK/STAT/FULL を実行したときの表示例 (VAX システムの例)

```
task statistics
    Total context switches:             0
    Number of existing threads:         0
    Total threads created:              0
DBG>
```

16.4.2 Ada タスクのタスク情報の表示

SHOW TASK/ALL コマンドでは，プログラム内に現存しているすべてのタスクの情報，つまりタスク自体が作成されてからその親がまだ終了していないすべてのタスクの情報が表示されます (Example 16–6 を参照)。

Example 16–6 Ada タスクに対して SHOW TASK/ALL を実行したときの表示例

```
 1        2    3     4        5             6
 task id pri hold state   substate     task object
* %TASK 1  7       RUN               SHARE$ADARTL+130428
  %TASK 2  7  HOLD SUSP  Accept       TASK_EXAMPLE.MOTHER+4
  %TASK 4  7       SUSP  Entry call   TASK_EXAMPLE.FATHER_TYPE$TASK_BODY.CHILD+4
  %TASK 3  6       READY             TASK_EXAMPLE.MOTHER)
DBG>
```

次の番号は， Example 16–6 の番号に対応しています。

1 タスク ID (第 16.3.3 項を参照)。アクティブ・タスクには左端の欄にアスタリスク(*)が付く。

2 タスク優先順位。 Ada の優先順位は 0 ～ 15。

VAX プロセッサで作成されるタスクは省略時の設定である優先順位 7 が割り当てられる。ただし，それ以外の優先順位がプラグマ PRIORITY で指定されている場合は，その優先順位が割り当てられる。

Alpha プロセッサで作成されるタスクは，タイム・スライス機能が使用不能ならば，省略時の設定である優先順位 7 が割り当てられる。タイム・スライス機能が使用可能ならば，プラグマ PRIORITY が指定されていないかぎり，適当な中間の値が優先順位として割り当てられる。

3 タスクが SET TASK/HOLD コマンドによって保留中かどうかを示す。タスクの保留については，第 16.5.1 項を参照。タスクの保留には，プログラムの以降の実行が許可されていれば，タスクの状態遷移の制限が伴う。

4 タスクの現在の状態 (表 16–3 を参照)。 RUN (RUNNING) 状態のタスクがアクティブ・タスク。表 16–3 には，プログラムの実行中に移る可能性のある状態が示されている。

5 タスクの現在の副次状態。副次状態は，あるタスク状態を引き起こした原因を推定するのに役立つ。表 16–5 を参照。

6 タスク・オブジェクトのデバッガのパス名，またはデバッガがタスク・オブジェクトをシンボル化できない場合はタスク・オブジェクトのアドレス。

表 16–5 Ada タスクの副次状態

タスクの副次状態	説明
Abnormal	タスクは強制終了された。
Accept	タスクは select 文の外の accept 文で待っている。
Activating	タスクはその宣言部分を作成中である。
Activating tasks	タスクは自分が作成した各タスクが活動を終了するのを待っている。
Completed [abn]	タスクは abort 文のために打ち切られたが，まだ終了していない。Ada でいう打ち切られたタスクとは，その end 文で各依存タスクを待っているタスクである。それらの依存タスクが終了すると，タスクの状態は Terminated に変わる。
Completed [exc]	タスクは処理されない例外[1]のために打ち切られたが，まだ終了していない。 Ada でいう打ち切られたタスクとは，その end 文で各依存タスクを待っているタスクである。それらの依存タスクが終了すると，タスクの状態は Terminated に変わる。
Completed	タスクは打ち切られている。 abort 文は実行されず，処理されない例外[1]は起こっていない。
Delay	タスクは delay 文で待っている。
Dependents	タスクは各依存タスクの終了を待っている。
Dependents [exc]	タスクは処理されない例外[1]の通知を依存タスクが許すのを待っている。
Entry call	タスクはそのエントリ呼び出しが受け付けられるのを待っている。
Invalid state	Ada の実行時ライブラリにエラーがある。
I/O または AST	タスクは入出力 (I/O) の終了かなんらかの AST を待っている。
Not yet activated	タスクは，自分を作成したタスクによって起動されるのを待っている。
Select or delay	タスクは遅延という代替方法が可能な select 文で待っている。
Select or terminate	タスクは終了という代替方法が可能な select 文で待っている。
Select	タスクは no else，遅延，終了のいずれも不可能な select 文で待っている。
Shared resource	タスクは内部共用リソースを待っている。
Terminated [abn]	タスクは abort 文により終了した。
Terminated [exc]	タスクは処理されない例外[1]により終了した。
Terminated	タスクは正常終了した。
Timed entry call	タスクは時限エントリ呼び出しをして待っている。

[1]処理されない例外とは，現在のフレーム内にそのためのハンドラがない例外，またはハンドラがあって raise 文を実行しその例外を外部有効範囲に通知する例外のことです。

図 16–1 はタスク・スタックを示します。

SHOW TASK/FULL コマンドでは，選択した各タスクについての詳細情報が表示されます。 Example 16–7 には， Ada タスク例に対してこのコマンドを実行したときの出力が示されています。

Example 16–7 ADA タスクに対して SHOW TASK/FULL を実行したときの表示例

```
1   task id     pri hold state   substate          task object
  * %TASK 2      7       RUN                    TASK_EXAMPLE.MOTHER+4

2         Waiting entry callers:
            Waiters for entry BOGUS:
                %TASK 4, type: CHILD

3       Task type:      FATHER_TYPE
        Created at PC:  TASK_EXAMPLE.%LINE 14+22
        Parent task:    %TASK 1
        Start PC:       TASK_EXAMPLE.FATHER_TYPE$TASK_BODY
4       Task control block:          5  Stack storage (bytes):
          Task value:   490816             RESERVED_BYTES:     10640
          Entries:      3                  TOP_GUARD_SIZE:      5120
          Size:         1488               STORAGE_SIZE:       30720
6       Stack addresses:                 Bytes in use:         456
          Top address:  001EB600
          Base address: 001F2DFC       7  Total storage:      47968
DBG>
```

次の番号は， Example 16–7 の番号に対応しています。

1 タスク情報の種別を示す。

2 ランデブ情報。呼び出しタスクの場合は，そのタスクが登録されているキューの各エントリが一覧表示される。呼び出されるタスクの場合は，実行されるランデブの種類が表示されるほか，そのタスクを待つキューのどれかのエントリに現在登録されている呼び出しタスクが一覧表示される。

3 タスクのコンテキスト情報。

4 タスク制御ブロック情報。「Task value」の右の 16 進数は，タスク制御ブロックのアドレスである。

5 スタック記憶域の情報。

- RESERVED_BYTES は， Ada の実行時ライブラリによってスタック・オーバフロー処理のために割り当てられた記憶域サイズである。
- TOP_GUARD_SIZE は，タスク実行中の記憶域オーバフローを防ぐために用意された記憶保護領域ページのサイズである。

  VAX システムでは，保護領域ページとして割り当てるバイト数は， Ada プラグマの TASK_STORAGE と MAIN_STORAGE に指定できる。デバッガにより表示される値は，割り当てられたバイト数である。プラグマの値は，ページ数が整数になるように，必要に応じて切り上げられる。これらのプラグマと上部記憶保護領域についての詳しい説明は， Ada のマニュアルを参照。

  Alpha システムでは，保護領域ページとして割り当てるバイト数は， Ada プラグマの TASK_STORAGE に指定できる。デバッガにより表示される値は，割り当てられたバイト数である。プラグマの値は，ページ数が整数になるよう

に，必要に応じて切り上げられる。これらのプラグマと上部記憶保護領域についての詳しい説明は， Ada のマニュアルを参照。

- STORAGE_SIZE は，タスクの起動用に割り当てられた記憶域のサイズである。

  VAX システムでは，割り当てるバイト数は， T'STORAGE_SIZE 表現句か Ada プラグマの MAIN_STORAGE に指定する。デバッガにより表示される値は，割り当てられたバイト数である。指定した値は，ページ数が整数になるように，必要に応じて切上げられる。この表現句とプラグマおよびタスク起動用 (作業用) 記憶域についての詳しい説明は， Ada のマニュアルを参照。

  AXP システムでは，割り当てるバイト数は， T'STORAGE_SIZE 表現句で指定する。デバッガにより表示される値は，割り当てられたバイト数である。指定した値は，ページ数が整数になるように，必要に応じて切上げられる。この表現句とプラグマおよびタスク起動用 (作業用) 記憶域についての詳しい説明は， Ada のマニュアルを参照。

- "Bytes in use:"は，スタックに現在割り当てられているバイト数である。

6 タスク・スタックのスタック・アドレス。

7 タスクが使用する記憶域の合計。タスク制御ブロックサイズ，予約領域バイト数，上部保護領域サイズ，および記憶域サイズの合計である。

SHOW TASK/STATISTICS コマンドでは，プログラム内のすべてのタスクについての統計情報が報告されます。 Example 16–8 は， VAX プロセッサで Ada タスキング・プログラム例に対して SHOW TASK/STATISTICS/FULL コマンドを実行したときの出力です。この情報により，プログラムの性能を測定できます。スケジューリング (コンテキスト・スイッチとも呼ぶ) の総数が多いほど，タスキング・オーバヘッドは大きくなります。

Example 16–8 Ada タスクに対して SHOW TASK/STATISTICS/FULL を実行したときの表示例 (VAX プロセッサの例)

```
task statistics
    Entry calls       = 4       Accepts = 1     Selects = 2
    Tasks activated   = 3       Tasks terminated   = 0
    ASTs delivered    = 4       Hibernations       = 0
    Total schedulings = 15
        Due to readying a higher priority task = 1
        Due to task activations                = 3
        Due to suspended entry calls           = 4
        Due to suspended accepts               = 1
        Due to suspended selects               = 2
        Due to waiting for a DELAY             = 0
```

(次ページに続く)

Example 16–8 (続き) Ada タスクに対して SHOW TASK/STATISTICS/FULL を実行したときの表示例 (VAX プロセッサの例)

```
        Due to scope exit awaiting dependents  = 0
        Due to exception awaiting dependents   = 0
        Due to waiting for I/O to complete     = 0
        Due to delivery of an AST              = 4
        Due to task terminations               = 0
        Due to shared resource lock contention = 0
DBG>
```

16.5 タスク特性の変更

デバッグ中にタスクの特性やタスキング環境を変更するには，次表の SET TASK コマンドを使用します。

コマンド	機能
SET TASK/ACTIVE	指定されたタスクをアクティブ・タスクにする。 POSIX Threads (OpenVMS Alpha システム， VAX システム，または Integrity システム) でも Ada (OpenVMS Alpha システムおよび Integrity システム) でも動作しない (第 16.3.1 項を参照)。
SET TASK/VISIBLE	指定されたタスクを可視タスクにする (第 16.3.1 項を参照)。
SET TASK/ABORT	次に可能な機会にタスクを終了するように要求する。具体的な効果はそのときのイベント機能により異なる (言語依存)。 Ada タスクの場合，これは abort 文の実行と同じである。
SET TASK/PRIORITY	タスクの優先順位を設定する。具体的な効果はそのときのイベント機能により異なる (言語依存)。
SET TASK/RESTORE	タスクの優先順位を復元する。具体的な効果はそのときのイベント機能により異なる (言語依存)。
SET TASK/[NO]HOLD	タスク・スイッチを制御する。タスク状態の遷移については第 16.5.1 項を参照。
SET TASK/TIME_SLICE	タイム・スライス値を制御するか，タイム・スライス機能を禁止する。VAX Adaのみでサポートされ， POSIX Threads ではサポートされていない (第 16.5.2 項を参照)。

詳細については， SET TASK コマンドの説明を参照してください。

16.5.1 タスクの保留によるタスク・スイッチの制御

タスク・スイッチにより，プログラムのデバッグが複雑になります。 SET TASK /HOLD コマンドを使用してタスクを保留すれば，あとでそのプログラムが実行可能になったときにそのタスクが移る状態を制限できます。

保留されたタスクは RUNNING 以外のいずれかの状態に移ります。しかし，必要であれば SET TASK/ACTIVE コマンドを使用して保留されたタスクを RUNNING 状態に移すことも可能です。

SET TASK/HOLD/ALL コマンドではアクティブ・タスク以外のすべてのタスクの状態が凍結されます。このコマンドを SET TASK/ACTIVE コマンドと併用すれば，指定した 1 つか複数のタスクの動作を観察できます。そのためには， STEP コマンドまたは GO コマンドによってそのアクティブ・タスクを実行し， SET TASK/ACTIVE コマンドを使用して実行を別のタスクに切り替えます。次に例を示します。

```
DBG> SET TASK/HOLD/ALL
DBG> SET TASK/ACTIVE %TASK 1
DBG> GO
   .
   .
   .
DBG> SET TASK/ACTIVE %TASK 3
DBG> STEP
   .
   .
   .
```

タスクを保留する必要がなくなったら， SET TASK/NOHOLD コマンドを使用します。

16.5.2 タイム・スライス機能を使用するプログラムのデバッグ (VAX のみ)

タイム・スライス機能を使用するタスキング・プログラムのデバッグは複雑です。タイム・スライス機能によってタスクの相対的な動作が非同期になるからです。タイム・スライス機能を使用しない場合，タスクの実行はタスクの優先順位だけで決まります。タスク・スイッチは予測可能なので，実行のたびにプログラムの同じ動作を繰り返すことができます。タイム・スライス機能を使用する場合もタスクの優先順位によってタスク・スイッチが行われますが，指定期間は同じ優先順位の複数のタスクが交互に実行されます。したがって，タイム・スライス機能があれば各タスクは互いにいっそう独立して実行されるようになります。そのため，タイム・スライス機能を使用するプログラムは，実行するたびに同じ動作を繰り返さないことがあります。

SET TASK/TIME_SLICE=*t*コマンド (VAX Adaのみでサポート) を使用すれば，新しいタイム・スライスを指定したり， SET TASK/TIME_SLICE=0.0 と指定してタイム・スライス機能を禁止したりできます。したがって，タスキング・プログラムの実行を調整したり，タスクの実行順序に依存する問題を診断したりできます。

注意

SET TASK/TIME_SLICE コマンドは，VAX Adaのみでサポートされており，POSIX Threads を使用したVAX Adaではサポートされていません。サポートされていないコンテキストでコマンドを使用する場合，次のエラー・メッセージが出されます。

```
%DEBUG-E-NOTIMSLI, time slice modification
     not supported by this event facility
```

タイム・スライス機能とデバッガのウォッチポイントの実行とは，互いに影響を与えることに注意してください。ウォッチポイントを設定すると，タイム・スライス間隔値はデバッガにより自動的に 10.0 秒に増加されることがあります。タイム・スライス速度を遅くすることにより，発生を防げる問題があるからです。

16.6 実行の制御とモニタ

次の各項では，次の各機能の実行方法について説明します。

- タスク依存およびタスク非依存のイベントポイント (ブレークポイント，トレースポイントなど) を設定する。
- POSIX Threads 固有のタスク記憶位置にブレークポイントとトレースポイントを設定する。
- Ada 固有のタスク記憶位置にブレークポイントとトレースポイントを設定する。
- SET BREAK/EVENT コマンドまたは SET TRACE/EVENT コマンドを使用してタスク・イベントをモニタする。

16.6.1 タスク依存およびタスク非依存のデバッガ・イベントポイントの設定

イベントポイントとは，デバッガに制御を戻すために使用できるイベントです。ブレークポイント，トレースポイント，ウォッチポイント，および STEP コマンドの終了はいずれもイベントポイントです。

タスク非依存イベントポイントは，プログラム内のいずれかのタスクの実行によって起動されます。そのイベントポイントの設定時にどのタスクがアクティブなのかは関係ありません。タスク非依存イベントポイントの指定には，行番号や名前などのアドレス式を使用するのが普通です。ウォッチポイントはすべて，タスク非依存イベントポイントです。次にタスク非依存イベントポイントの設定例を示します。

```
DBG> SET BREAK COUNTER
DBG> SET BREAK/NOSOURCE %LINE 55, CHILD$TASK_BODY
DBG> SET WATCH/AFTER=3 KEEP_COUNT
```

タスク依存イベントポイントは，コマンド入力時にアクティブなタスクにしか設定できません。タスク依存イベントポイントは，その同じタスクがアクティブなときにしか起動されません。たとえば， STEP/LINE コマンドはタスク依存イベントポイントです。その他のタスクが同じソース行を実行してもそのイベントは起動されないことがあります。

次の修飾子を指定して SET BREAK, SET TRACE，または STEP コマンドを実行すれば，タスク依存イベントポイントが設定されます。

/BRANCH
/CALL

/INSTRUCTION
/LINE
/RETURN
/VECTOR_INSTRUCTION (VAX 専用)

これらのコマンドでこれらの修飾子を使用しないで設定するイベントポイントおよび SET WATCH コマンドを使用して設定するイベントポイントは，タスク非依存になります。次にタスク依存イベントポイントの設定例を示します。

```
DBG> SET BREAK/INSTRUCTION
DBG> SET TRACE/INSTRUCTION/SILENT DO (EXAMINE KEEP_COUNT)
DBG> STEP/CALL/NOSOURCE
```

通常はタスク非依存のイベントポイントに条件を設定して，タスク依存にすることができます。次に例を示します。

```
DBG> SET BREAK %LINE 11 WHEN (%ACTIVE_TASK=FATHER)
```

16.6.2 POSIX Threads タスキング構造へのブレークポイントの設定

ブレークポイントをスレッド起動ルーチンに設定することは可能です。そのようなブレークポイントは，起動ルーチンの実行開始直前に検出されます。 Example 16–1 の場合は，この型のブレークポイントはたとえば次のように設定します。

```
DBG> SET BREAK worker_routine
```

Ada タスクの場合とは異なり POSIX Threads タスクの場合は名前で本体を指定することはできませんが，起動ルーチンは似ています。

SET BREAK コマンドに WHEN 句を指定すれば，特定のスレッドの実行開始時点を確実にとらえることができます。次に例を示します。

```
DBG> SET BREAK worker_routine -
_DBG> WHEN (%CALLER_TASK = %TASK 4)
```

Example 16–1 の場合，この条件付きブレークポイントは 2 番目のワーカ・スレッドがその起動ルーチンを開始するときに検出されます。

ブレークポイントの設定に適するその他の箇所には，条件待ち，結合，およびミューテクスのロックの直前と直後があります。そのようなブレークポイントの設定には，行番号かルーチン名を指定します。

16.6.3 Ada タスク本体，エントリ呼び出し，および accept 文へのブレークポイントの設定

タスク本体にブレークポイントを設定するには，次のどちらかの構文に従ってタスク本体を指定します (第 16.3.2 項を参照)。

task-type-identifier$TASK_BODY

task-identifier$TASK_BODY

たとえば，次のコマンドではタスク CHILD の本体にブレークポイントが設定されます。そのブレークポイントは，タスクの宣言部分の作成 (タスクのアクティベーションとも呼ぶ) の直前に検出されます。

```
DBG> SET BREAK CHILD$TASK_BODY
```

CHILD$TASK_BODY は，タスクが最初に実行する命令が置かれている箇所の名前です。ブレークポイントをある命令に設定するのは意味があることなので，この名前を指定します。しかし， SET BREAK コマンドにタスク・オブジェクトの名前 (たとえば， CHILD) を指定してはなりません。タスク・オブジェクト名は，データ項目のアドレス (32 ビットのタスク値) を示します。整数オブジェクトにブレークポイントを設定するのは間違いであるように，タスク・オブジェクト名にブレークポイントを設定するのも間違いです。

エントリ呼び出し文と accept 文にブレークポイントかトレースポイントを設定すれば，呼ぶ側，呼ばれる側のタスクの実行をモニタできます。

注意

Ada タスクのエントリ呼び出しは，サブプログラム呼び出しと同じではありません。タスク・エントリ呼び出しはキューに登録されるので，すぐ実行されるとは限らないからです。 STEP コマンドを使用して実行をタスク・エントリ呼び出しに移しても，期待通りの結果が得られないことがあります。

ブレークポイントやトレースポイントの設定に適する箇所は， accept 文とその前後に数箇所あります。たとえば， RENDEZVOUS という同じエントリに 2 つの accept 文が使用されている次のプログラム・セグメントについて考えてみます。

```
 8   task body TWO_ACCEPTS is
 9   begin
10      for I in 1..2 loop
11         select
12            accept RENDEZVOUS do
13               PUT_LINE("This is the first accept statement");
14            end RENDEZVOUS;
15         or
16            terminate;
17         end select;
18      end loop;
```

```
19      accept RENDEZVOUS do
20         PUT_LINE("This is the second accept statement");
21      end RENDEZVOUS;
22   end TWO_ACCEPTS;
```

この例では，次の箇所にブレークポイントかトレースポイントを設定できます。

- accept 文の先頭 (行 12 か行 19)。ここにブレークポイントかトレースポイントを設定すれば，実行が accept 文の先頭に達してランデブが実際に起こる前に，タスクの受け入れが中断すると思われる箇所をモニタできる。
- accept 文の本体 (一連の文) の先頭 (行 13 か行 20)。ここにブレークポイントかトレースポイントを設定すれば，ランデブが始まる箇所，つまり実際に accept 文の実行が始まる箇所をモニタできる。
- accept 文の最後 (行 14 か行 21)。ここにブレークポイントかトレースポイントを設定すれば，ランデブが終了し，実行が呼び出し側タスクに切り替えられるところをモニタできる。

accept 文かその前後にブレークポイントかトレースポイントを設定するためには，その行番号を指定します。たとえば，次のコマンドでは，前の例の最初の accept 文の先頭と本体にブレークポイントが設定されます。

```
DBG> SET BREAK %LINE 12, %LINE 13
```

accept 文の本体にブレークポイントかトレースポイントを設定するには，エントリ名も使用できます。その展開された名前を指定して，そのエントリが宣言されているタスク本体を示します。次に例を示します。

```
DBG> SET BREAK TWO_ACCEPTS$TASK_BODY.RENDEZVOUS
```

同じエントリに 2 つ以上の accept 文がある場合，デバッガはそのエントリをオーバーロードされた名前として扱います。デバッガからそのシンボルがオーバーロードされていることを示すメッセージが発行されるので，ユーザは SHOW SYMBOL コマンドを使用して，デバッガによって割り当てられたオーバーロードされた名前を表示しなければなりません。次に例を示します。

```
DBG> SHOW SYMBOL RENDEZVOUS
overloaded symbol TEST.TWO_ACCEPTS$TASK_BODY.RENDEZVOUS
  overloaded instance TEST.TWO_ACCEPTS$TASK_BODY.RENDEZVOUS__1
  overloaded instance TEST.TWO_ACCEPTS$TASK_BODY.RENDEZVOUS__2
```

オーバーロードされた名前の後ろには， 2 つの下線と整数が続きます。オーバーロードされた名前についての詳しい説明は，デバッガのオンライン・ヘルプを参照してください (Help Language_Support Ada を入力します)。

オーバーロードされた名前のどちらが特定の accept 文に対応しているかを知るためには， EXAMINE/SOURCE コマンドを使用します。次に例を示します。

```
DBG> EXAMINE/SOURCE TWO_ACCEPTS$TASK_BODY.RENDEZVOUS__1
module TEST_ACCEPTS
    12:       accept RENDEZVOUS do
DBG> EXAMINE/SOURCE TWO_ACCEPTS$TASK_BODY.RENDEZVOUS__2
module TEST_ACCEPTS
    19:       accept RENDEZVOUS do
```

次の例でブレークポイントが検出されると，呼び出しタスクが評価されます。シンボル%CALLER_TASK についての詳しい説明は，第 16.3.4 項を参照してください。

```
DBG> SET BREAK TWO_ACCEPTS$TASK_BODY.RENDEZVOUS__2 -
_DBG> DO (EVALUATE %CALLER_TASK)
```

次のブレークポイントが検出されるのは，呼び出しタスクが%TASK 2 のときだけです。

```
DBG> SET BREAK TWO_ACCEPTS$TASK_BODY.RENDEZVOUS__2 -
_DBG> WHEN (%CALLER_TASK = %TASK 2)
```

呼び出しタスクが同じ accept 文に 2 回以上エントリ呼び出しを行う場合は， SHOW TASK/CALLS コマンドを使用することにより，そのエントリ呼び出しが実行されたソース行を表示できます。たとえば，次のように指定します。

```
DBG> SET BREAK TWO_ACCEPTS$TASK_BODY.RENDEZVOUS__2 -
_DBG> DO (SHOW TASK/CALLS %CALLER_TASK)
```

16.6.4 タスク・イベントのモニタ

SET BREAK/EVENT コマンドと SET TRACE/EVENT コマンドを使用すれば，タスク・イベントと例外イベントによって検出されるブレークポイントおよびトレースポイントを設定できます。たとえば，次のコマンドで設定するトレースポイントは，タスク CHILD か%TASK 2 が RUN 状態に移るときに検出されます。

```
DBG> SET TRACE/EVENT=RUN CHILD,%TASK 2
```

あるイベントの結果，ブレークポイントかトレースポイントが検出されると，デバッガはそのイベントを識別し，追加情報を与えます。

次の各表に， SET BREAK/EVENT コマンドと SET TRACE/EVENT コマンドに\指定できるイベント名キーワードの一覧を示します。

- 表 16–6 に，すべてのタスクに共通する一般的な言語非依存イベントの一覧を示す。
- 表 16–7 に， POSIX Threads タスクに固有のイベントの一覧を示す。

- 表 16–8 に， Ada タスクに固有のイベントの一覧を示す。

表 16–6 下位レベルの汎用タスクのスケジューリング・イベント

イベント名	機能
RUN	タスクが実行する直前に検出される。
PREEMPTED	RUN 状態にあるタスクが強制的に READY 状態に移される直前に検出される (表 16–3 を参照)。
ACTIVATING	タスクが実行を開始する直前に検出される。
SUSPENDED	タスクが中断される直前に検出される。

表 16–7 POSIX Threads 依存イベント

イベント名	機能
HANDLED	ある TRY ブロックで例外が処理される直前に検出される。
TERMINATED	タスクが終了する直前に検出される (警告や例外による場合を含む)。
EXCEPTION_TERMINATED	タスクが例外によって終了する直前に検出される。
FORCED_TERM	タスクが警告処理によって終了する直前に検出される。

表 16–8 Ada 固有のイベント

イベント名	機能
HANDLED	例外が Ada の例外ハンドラ (その他のハンドラを含む) によって処理される直前に検出される。
HANDLED_OTHERS	例外がその他の Ada 例外ハンドラによって処理される直前にだけ検出される。
RENDEZVOUS_EXCEPTION	例外がランデブの外に通知される直前に検出される。
DEPENDENTS_EXCEPTION	例外によってタスクがある有効範囲内の依存タスクを待つ直前に検出される (未処理例外[1]を含む。したがって，Ada の実行時ライブラリ内部の特殊な例外を含む。詳しくは， Ada のマニュアルを参照)。デッドロックの直前のことが多い。
TERMINATED	タスクの終了直前に検出される。正常終了のとき， abort 文によるとき，および例外によるときを含む。
EXCEPTION_TERMINATED	処理されない例外[1]によってタスクが終了する直前に検出される。
ABORT_TERMINATED	abort 文によってタスクが終了する直前に検出される。

[1]未処理例外とは，現在のフレーム内にそのためのハンドラがない例外，またはハンドラがあって raise 文を実行しその例外を外部有効範囲に通知する例外のことです。

上記の各表には，例外関係のイベントも含めてあります。次の各段落では，タスク・イベントについてだけ説明しています。例外イベントについての詳しい説明は，デバッガのオンライン・ヘルプを参照してください (type Help Language_Support Adaを入力します)。

イベント名キーワードは，一意性を損わない文字数まで短縮できます。

SET BREAK/EVENT コマンドまたは SET TRACE/EVENT コマンドに指定できるイベント名キーワードは，そのときのイベント機能がタスク・イベント時に THREADS なのか ADA なのかで異なります。プログラムをデバッガの制御下に置くと，適切なイベント機能が自動的に設定されます。 SHOW EVENT_FACILITY コマンドでは，現在設定されている機能が示され，その機能に指定できるイベント名キーワード (汎用イベントのキーワードを含む) の一覧が表示されます。

以後，いくつかの例によって/EVENT 修飾子の使用法を示します。

```
DBG> SET BREAK/EVENT=PREEMPTED
DBG> GO
break on THREADS event PREEMPTED
  Task %TASK 4 is getting preempted by %TASK 3
    .
    .
    .
DBG> SET BREAK/EVENT=SUSPENDED
DBG> GO
break on THREADS event SUSPENDED
  Task %TASK 1 is about to be suspended
    .
    .
    .
DBG> SET BREAK/EVENT=TERMINATED
DBG> GO
break on THREADS event TERMINATED
  Task %TASK 4 is terminating normally
DBG>
```

プログラムをデバッガの制御下に置くと，次の定義済みイベント・ブレークポイントが自動的に設定されます。

- EXCEPTION_TERMINATED イベント・ブレークポイントは， POSIX Threads ルーチンを呼び出すプログラムでは定義済みである。
- EXCEPTION_TERMINATED と DEPENDENTS_EXCEPTION の各イベント・ブレークポイントは， Ada プログラムか Ada ルーチンを呼び出すプログラムでは定義済みである。

以後， Ada での定義済みおよびその他の型のイベント・ブレークポイント例を示します。

EXCEPTION_TERMINATED イベントの例

EXCEPTION_TERMINATED イベントが起動されるときには，通常，予期しないプログラム・エラーが示されます。次に例を示します。

```
 . . .
break on ADA event EXCEPTION_TERMINATED
  Task %TASK 2 is terminating because of an exception
    %ADA-F-EXCCOP, Exception was copied at a "raise;" or "accept"
    -ADA-F-EXCEPTION, Exception SOME_ERROR
    -ADA-F-EXCRAIPRI, Exception raised prior to PC = 00000B61
DBG>
```

DEPENDENTS_EXCEPTION **イベントの例** (Ada)

Ada プログラムの場合， DEPENDENTS_EXCEPTION イベントが起こると，そのあとにデッドロックが続くことがよくあります。次に例を示します。

```
 . . .
break on ADA event DEPENDENTS_EXCEPTION
  Task %TASK 2 may await dependent tasks because of this exception:
    %ADA-F-EXCCOP, Exception was copied at a "raise;" or "accept"
    -ADA-F-EXCEPTION, Exception SOME_ERROR
    -ADA-F-EXCRAIPRI, Exception raised prior to PC = 00000B61
DBG>
```

RENDEZVOUS_EXCEPTION **イベントの例** (Ada)

Ada プログラムの場合， RENDEZVOUS_EXCEPTION イベントを指定しておけば，制御がランデブを離れる前に (例外情報が呼び出しタスクにコピーされて失われる前に) その例外を見ることができます。次に例を示します。

```
 . . .
break on ADA event RENDEZVOUS_EXCEPTION
  Exception is propagating out of a rendezvous in task %TASK 2
    %ADA-F-CONSTRAINT_ERRO, CONSTRAINT_ERROR
    -ADA-I-EXCRAIPRI, Exception raised prior to PC = 00000BA6
DBG>
```

/EVENT 修飾子によって設定されたブレークポイントまたはトレースポイントを取り消すには， CANCEL BREAK/EVENT または CANCEL TRACE/EVENT コマンドを使用します。 SET コマンドのときと全く同じように， CANCEL コマンドにイベント修飾子とオプションのタスク式を指定します。ただし， WHEN 句や DO 句は指定しません。

タスキング・プログラム用のデバッガ初期化ファイルに，イベント・ブレークポイントとトレースポイントを設定したいことがあります。次に例を示します。

```
SET BREAK/EVENT=ACTIVATING
SET BREAK/EVENT=HANDLED DO (SHOW CALLS)
SET BREAK/EVENT=ABORT_TERMINATED DO (SHOW CALLS)
SET BREAK/EVENT=EXCEPTION_TERM DO (SHOW CALLS)
SET BREAK/EVENT=TERMINATED
```

16.7 タスク・デバッグについての補足

次の各項では，タスク・デバッグに関係する次の補足事項について説明します。

- デッドロック
- 自動スタック・チェック

- Ctrl/Y の使用法

16.7.1 デッドロック状態になるプログラムのデバッグ

デッドロックとは，あるタスク・グループ内の各タスクが中断されて，そのグループ内のどれか別のタスクが実行されないかぎり，グループ内のどのタスクも実行を再開できないエラー状態のことです。デッドロックは，タスキング・プログラムでよく起こるエラーです。 WHILE 文を使用するプログラムでは無限ループがよく起こるエラーであるのと同様です。

デッドロックは簡単に検出できます。この状態のプログラムは実行を中断しているように見えます。デバッガの制御下で実行しているプログラムがデッドロックを起こしたときには， Ctrl/C を押します。そのデッドロックが中断され，デバッガのプロンプトが表示されます。

通常は， SHOW TASK/ALL コマンド (第 16.4 節を参照) か SHOW TASK /STATE=SUSPENDED コマンドが役立ちます。プログラム内の中断しているタスクとその理由が示されるからです。画面モードでデバッグしているときは， SET TASK/VISIBLE %NEXT_TASK コマンドがたいへん便利です。これを使用すれば，すべてのタスクを調べ，各タスクが実行しているコード (実行が停止しているコードを含む) を表示できるからです。

SHOW TASK/FULL コマンドでは，ランデブ情報，エントリ呼び出し情報，エントリ索引値などのタスク状態についての詳しい情報が得られます。 SET BREAK /EVENT コマンドや SET TRACE/EVENT コマンド (第 16.6.4 項を参照) では，デッドロックを引き起こす可能性がある記憶位置かその近くにブレークポイントやトレースポイントを設定できます。 SET TASK/PRIORITY コマンドと SET TASK /RESTORE コマンドでは，全く実行されない低優先順位のタスクがデッドロックを引き起こしているのかどうかが分かります。

表 16–9 に，さまざまなタスクのデッドロック状態と，その原因の診断に役立つと思われるデバッガ・コマンドの一覧を示します。

表 16–9 Ada タスクのデッドロック状態とそれを診断するためのデバッガ・コマンド

デッドロック状態	デバッガ・コマンド
自己呼び出しデッドロック (タスクが自分自身のエントリの 1 つを呼び出す)	SHOW TASK/ALL SHOW TASK/STATE=SUSPENDED SHOW TASK/FULL
循環呼び出しデッドロック (あるタスクが別のタスクを呼び出し，呼び出されたタスクが最初のタスクを呼び出す)	SHOW TASK/ALL SHOW TASK/STATE=SUSPENDED SHOW TASK/FULL

(次ページに続く)

表 16–9 (続き) Ada タスクのデッドロック状態とそれを診断するためのデバッガ・コマンド

デッドロック状態	デバッガ・コマンド
動的呼び出しデッドロック (一連のエントリ呼び出しが循環し，そのうちの少なくとも 1 つがループ内の時限または条件付きエントリ呼び出しである)	SHOW TASK/ALL SHOW TASK/STATE=SUSPENDED SHOW TASK/FULL
例外によって生じるデッドロック (例外によって，タスクがそのエントリ呼び出しの 1 つに応えられないまたは例外の通知が依存タスクを待たなければならない)	SHOW TASK/ALL SHOW TASK/STATE=SUSPENDED SHOW TASK/FULL SET BREAK/EVENT=DEPENDENTS_EXCEPTION (Ada プログラムの場合)
エントリ索引に対する実行時の計算間違い， when 条件，および select 文内の delay 文によるデッドロック	SHOW TASK/ALL SHOW TASK/STATE=SUSPENDED SHOW TASK/FULL EXAMINE
エントリが間違った順番で呼び出されたためのデッドロック	SHOW TASK/ALL SHOW TASK/STATE=SUSPENDED SHOW TASK/FULL
優先順位の低いタスクによって設定される変数をフラグとして使用するが，それより優先順位の高いタスクがいつもレディ状態のため優先順位の低いタスクが全く実行できなくて生じる，変数の変更待ちデッドロック	SHOW TASK/ALL SHOW TASK/STATE=SUSPENDED SHOW TASK/FULL SET TASK/PRIORITY SET TASK/RESTORE

16.7.2 デバッガによる自動スタック・チェック

タスキング・プログラムの場合，ある種の状況では検出されないスタック・オーバフローが起こり，予期しない処理が行われることがあります。タスク・スタック・オーバフローについての詳しい説明は， Ada か POSIX Threads のマニュアルを参照してください。デバッガによって自動的に次のスタック・チェックが行われるので，スタック・オーバフローの問題の原因を探り出すのに役立ちます。スタック・ポインタが境界を越えている場合は，エラー・メッセージが表示されます。

- STEP コマンドの実行後，またはブレークポイントの検出後に，アクティブ・タスクのスタック・チェックが行われる (第 16.6.1 項を参照)。 STEP または SET BREAKPOINT コマンドに/SILENT 修飾子を指定していると，このチェックは行われない。
- SHOW TASK コマンドによって状態が表示される各タスクについてスタック・チェックが行われる。したがって， SHOW TASK/ALL コマンドを実行すればすべてのタスクのスタックが自動的にチェックされる。

次のエラー・メッセージ例は，スタック・チェックによってエラーが見つかったときに発行されるものです。たとえスタックがまだオーバフローしていなくても，スタックの大半が使用し尽くされているときには，警告メッセージが発行されます。

```
warning: %TASK 2 has used up over 90% of its stack
  SP: 0011194C  Stack top at: 00111200  Remaining bytes: 1868

error: %TASK 2 has overflowed its stack
  SP: 0010E93C  Stack top at: 00111200  Remaining bytes: -10436

error: %TASK 2 has underflowed its stack
  SP: 7FF363A4  Stack base at: 001189FC  Stack top at: 00111200
```

スタックのオーバフローに続いてスタックのアンダフローが起こることがあります。その経過は次のとおりです。タスク・スタックがオーバフローしてスタック・ポインタが上部保護領域を指したままでいるとき，オペレーティング・システムは ACCVIO 状態をシグナル通知しようとします。しかし上部保護領域には ACCVIO のシグナル引数を書き込めないので，オペレーティング・システムはスタックの別の場所を探します。つまり，フレーム・ポインタとスタック・ポインタがメイン・プログラムのスタック領域の基底を指すようにしてシグナル引数を書き込み，プログラム・カウンタを変更してイメージを強制終了します。

このときタイム・スライス AST かその他の AST が発生すると，別のタスクの実行が再開し，しばらくの間は正常でないままプログラムが続行することがあります。スタックがオーバフローしたタスクは，メイン・プログラムのスタックを使用し，重ね書きすることがあります。デバッガのスタック・チェックは，このような状態を検出するのに役立ちます。オペレーティング・システムによってスタックの規定外の箇所に書き込みをされたタスク内の命令をステップ実行すれば，またはそのときに SHOW TASK/ALL コマンドを使用すれば，デバッガからスタック・アンダフロー・メッセージが発行されます。

16.7.3 Ada タスクをデバッグするときの Ctrl/Y の使用

デバッグ・セッション中にプログラムの実行やデバッガ・コマンドに割り込むには，Ctrl/C を押すのが望ましい方法です。 Ctrl/C を押せば制御はデバッガに戻りますが， Ctrl/Y を押せば制御は DCL レベルに戻ります。

Ctrl/Y を押してタスク・デバッグ・セッションに割り込むと， DEBUG コマンドを使用して DCL レベルでデバッガを開始するときに問題が生じることがあります。そのような場合は，メイン・プログラムのソース・コードの先頭に次の 2 行を挿入して，Ada の定義済みパッケージ CONTROL_C_INTERCEPTION の名前を指定してください。

```
with CONTROL_C_INTERCEPTION;
pragma ELABORATE(CONTROL_C_INTERCEPTION);
```

このパッケージについての詳しい説明は， Comapq Ada のマニュアルを参照してください。

第6部
付録

A

定義済みのキー機能

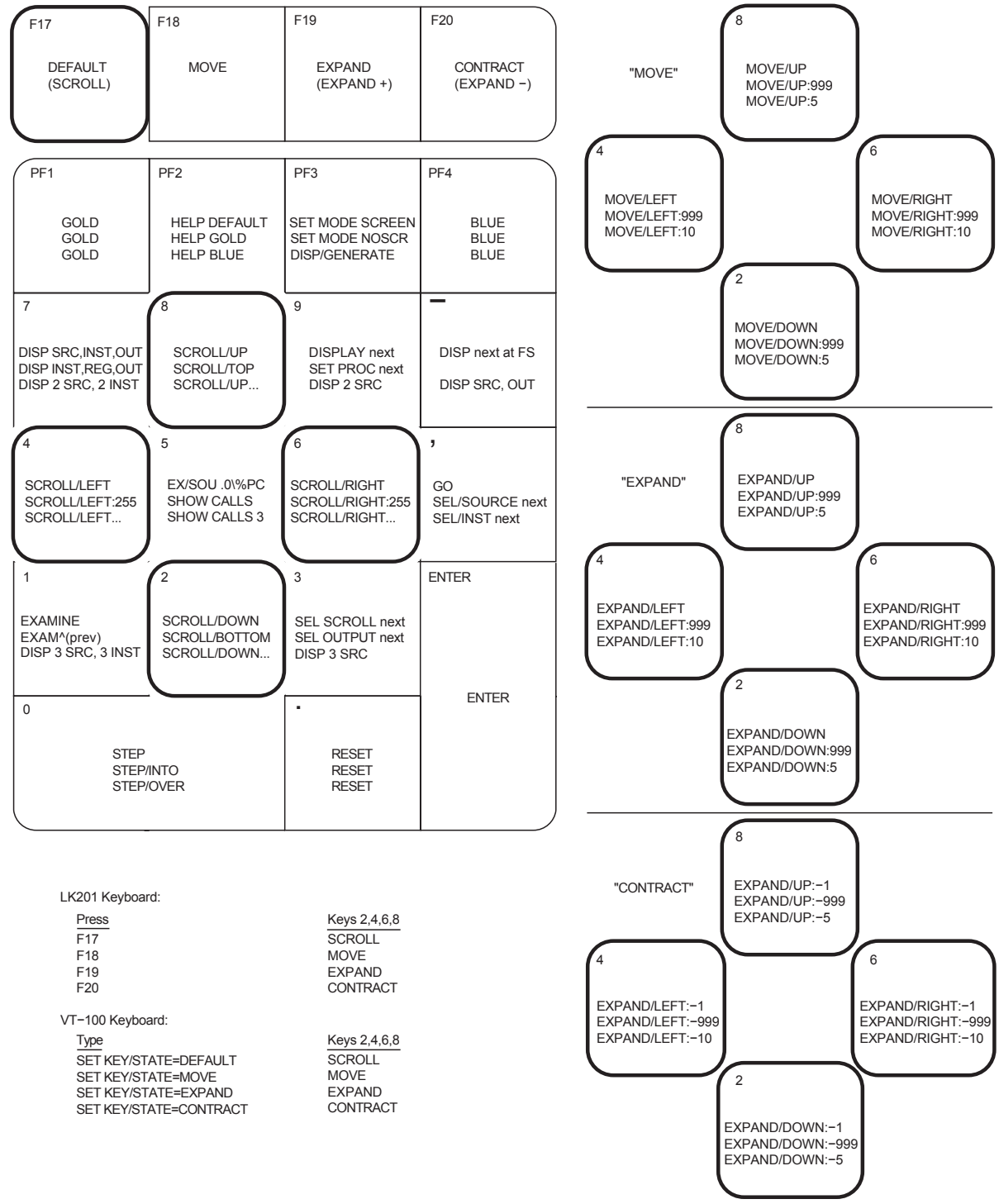
デバッガを起動する場合，定義済みの機能 (コマンド，コマンド・シーケンス，コマンド区切り文字) がメイン・キーパッドの右側にある数値キーパッドのキーに割り当てられています。これらのキーを使用すると，キーボードでコマンドを入力するよりも少ないキー入力でコマンドを入力できます。たとえば，キーパッド上で COMMA キー(,)を押せば， GO を入力してから Return キーを押すのと同じ機能を指定できます。 LK201 キーボードを持つ端末やワークステーションには， VT100 キーボードにはないプログラマブル・キー (たとえば，“Help”や“Remove”など) があります。またその中には，デバッガ機能が割り当てられているものもあります。

ファンクション・キーを使用するには，キーパッド・モードを使用可能 (SET MODE KEYPAD) にする必要があります。キーパッド・モードは，デバッガを起動するときに使用可能にします。デバッグするプログラムがキーパッドを使用しているため，キーパッド・モードを使用可能にしない方がよい場合は， SET MODE NOKEYPAD コマンドを入力してキーパッド・モードを禁止することができます。

デバッガを起動するときに定義済みであるキーパッド・キー機能について，図 A–1 にまとめてあります。また，すべてのキー定義の詳細な説明は，表 A–1，表 A–2，表 A–3，表 A–4 に示します。ほとんどのキーは，画面モードで画面表示を操作するのに使用されます。画面モードのコマンドを使用するには，まず PF3 キー (SET MODE SCREEN) を押して画面モードを使用可能にしなければなりません。また，画面モードで各ウィンドウの省略時のレイアウトを再作成するには，キーパッド・キー・シーケンス BLUE-MINUS (PF4 キーと MINUS キー(-)を順に押す) を入力します。

キーパッド・キーを使用してデバッガ・コマンドではなく数字を入力する場合は，SET MODE NOKEYPAD コマンドを入力します。

図 A–1 デバッガによって定義済みのキーパッド・キー機能 — コマンド・インタフェース



LK201 Keyboard:

Press	Keys 2,4,6,8
F17	SCROLL
F18	MOVE
F19	EXPAND
F20	CONTRACT

VT−100 Keyboard:

Type	Keys 2,4,6,8
SET KEY/STATE=DEFAULT	SCROLL
SET KEY/STATE=MOVE	MOVE
SET KEY/STATE=EXPAND	EXPAND
SET KEY/STATE=CONTRACT	CONTRACT

ZK−0956A−GE

A.1 機能 DEFAULT，GOLD，BLUE

1 つのキーにはたいてい次の 3 つの機能が定義されています。

- 特定のキーを押すことで， DEFAULT 機能を入力する。
- GOLD キーとも呼ばれる PF1 キーを押して離した後に，特定のキーを押すことで， GOLD 機能を入力する。
- BLUE キーとも呼ばれる PF4 キーを押して離した後に，特定のキーを押すことで， BLUE 機能を入力する。

図 A–1 の各キーの説明には， DEFAULT， GOLD， BLUE の各機能を上から順に表示します。たとえば，キーパッド・キー KP0 を押すと， STEP コマンド (DEFAULT 機能) が入力され， PF1， KP0 の順に押すと STEP/INTO コマンド (GOLD 機能) が入力され， PF4， KP0 の順に押すと STEP/OVER コマンド (BLUE 機能) が入力されます。

キーパッド・キーに割り当てられたコマンド・シーケンスは， KP2， KP4， KP6， KP8 に割り当てられた BLUE 機能以外は完結型で，ただちに実行されます。
KP2 などのような非完結型のコマンドには，図 A–1 で，コマンドの後ろに反復記号(. . .)をつけて区別してあります。コマンド入力を完結させるには，パラメータを指定してから Return キーを押します。たとえば， 12 行分をスクロール・ダウンするには，次のようにします。

1. PF4 キーを押す。
2. キーパッド・キー KP2 を押す。
3. キーパッドで:12 を入力する。
4. Return キーを押す。

A.2 LK201 キーボードに固有なキー定義

表 A–1 に， LK201 キーボードにだけあり， VT100 キーボードにはないキーを示します。また，そのキーごとに相当するコマンドと， LK201 キーボードを使用していない場合に代わりに使用できるキーパッド・キー (該当する場合) も示します。

表 A–1 LK201 キーボードに固有なキー定義

LK201 キー	起動されるコマンド・シーケンス	等価のキーパッド・キー
F17	SET KEY/STATE=DEFAULT	なし
F18	SET KEY/STATE=MOVE	なし

(次ページに続く)

表 A–1 (続き) LK201 キーボードに固有なキー定義

LK201 キー	起動されるコマンド・シーケンス	等価のキーパッド・キー
F19	SET KEY/STATE=EXPAND	なし
F20	SET KEY/STATE=CONTRACT	なし
Help	HELP KEYPAD SUMMARY	なし
Next Screen	SCROLL/DOWN	KP2
Prev Screen	SCROLL/UP	KP8
Remove	DISPLAY/REMOVE %CURSCROLL	なし
Select	SELECT/SCROLL %NEXTSCROLL	KP3

A.3 表示のスクロール，移動，拡大，縮小を行うキー

省略時の設定では，キーパッド・キー KP2， KP4， KP6， KP8 は，現在のスクロール・ディスプレイをスクロールします。これらのキーは，方向 (それぞれ，下，左，右，上) を制御します。また， F18， F19，または F20 を押すと，キーパッドをそれぞれ MOVE 状態， EXPAND 状態または CONTRACT 状態にすることができます。キーパッドが MOVE 状態にあると， KP2， KP4， KP6， KP8 は，現在のスクロール・ディスプレイをそれぞれ下，左，右，上へ移動できます。同様に EXPAND 状態と CONTRACT 状態にあると， 4 つのキーを使用して現在のスクロール・ディスプレイを拡大または縮小できます。図 A–1 と表 A–2 を参照してください。また， VT100 キーボード用の等価キーの定義については，あとで説明します。

ディスプレイのスクロール，移動，拡大，または縮小を行うには，次のようにします。

1. KP3 を必要なだけ繰り返し押して，ディスプレイ・リストから現在のスクロール・ディスプレイを選択する。
2. F17， F18， F19，または F20 を押して，キーパッドをそれぞれ DEFAULT (スクロール) 状態， MOVE 状態， EXPAND 状態， CONTRACT 状態にする。
3. KP2， KP4， KP6， KP8 を押して，必要な機能を実行する。スクロール量または移動量を制御するには， PF1 (GOLD) と PF4 (BLUE) を使用する。

表 A–2 キー状態を変更するキー

キー	機能
PF1	次に押すキーの GOLD 機能を起動する。
PF4	次に押すキーの BLUE 機能を起動する。

(次ページに続く)

表 A–2 (続き) キー状態を変更するキー

キー	機能
F17	キーパッドを DEFAULT 状態にし， KP2， KP4， KP6， KP8 のディスプレイ・スクロール機能を使用可能にする。デバッガを起動すると，キーパッドは DEFAULT 状態になる。
F18	キーパッドを MOVE 状態にし， KP2， KP4， KP6， KP8 のディスプレイ移動機能を使用可能にする。
F19	キーパッドを EXPAND 状態にし， KP2， KP4， KP6， KP8 のディスプレイ拡大機能を使用可能にする。
F20	キーパッドを CONTRACT 状態にし， KP2， KP4， KP6， KP8 のディスプレイ縮小機能を使用可能にする。

VT100 キーボードを使用している場合には， LK201 の F17 キーの効果を F20 にシミュレートできます。このためには，これらのキーの機能を他のキー・シーケンスに割り当てます。キーの割り当ては，デバッガ初期化ファイルなど，コマンド・プロシージャで実行できます (init_sec を参照)。次のコードには， GOLD-KP9 と BLUE-KP9 (現在は未定義) が F17 ～ F20 キーを巡回したときの効果をシミュレートできるようなキー割り当てが指定されています。これらのキー割り当てが有効な場合には， GOLD-KP9 を押すと，キーパッドは DEFAULT (スクロール) 状態になります。 BLUE-KP9 を押すと，キーパッドの状態は DEFAULT， MOVE， EXPAND， CONTRACT に順に切り換わります。

```
DEFINE/KEY/IF_STATE=(GOLD,MOVE_GOLD,EXPAND_GOLD,CONTRACT_GOLD)-
 /TERMINATE KP9 "SET KEY/STATE=DEFAULT/NOLOG"
DEFINE/KEY/IF_STATE=(BLUE)-
 /TERMINATE KP9 "SET KEY/STATE=MOVE/NOLOG"
DEFINE/KEY/IF_STATE=(MOVE_BLUE)-
 /TERMINATE KP9 "SET KEY/STATE=EXPAND/NOLOG"
DEFINE/KEY/IF_STATE=(EXPAND_BLUE)-
 /TERMINATE KP9 "SET KEY/STATE=CONTRACT/NOLOG"
DEFINE/KEY/IF_STATE=(CONTRACT_BLUE)-
 /TERMINATE KP9 "SET KEY/STATE=DEFAULT/NOLOG"
```

A.4 オンライン・キーパッド・キー図

Help キーや PF2 キーをそれだけ，または他のキーとともに押すと，キーパッド・キーのオンライン・ヘルプを利用できます。 (表 A–3 を参照してください。) また，SHOW KEY コマンドを使用してもキー定義を表示できます。

表 A–3 キーパッド図を表示するオンライン・ヘルプを起動するキー

キーまたは キー・シーケンス	起動されるコマンド・シーケンス	機能
Help	HELP KEYPAD SUMMARY	キーパッド・キー図を表示し，各キーの機能を要約する。
PF2	HELP KEYPAD DEFAULT	キーパッド・キー図とその DEFAULT 機能を表示する。
PF1-PF2	HELP KEYPAD GOLD	キーパッド・キー図とその GOLD 機能を表示する。
PF4-PF2	HELP KEYPAD BLUE	キーパッド・キー図とその BLUE 機能を表示する。
F18-PF2	HELP KEYPAD MOVE	キーパッド・キー図とその MOVE DEFAULT 機能を表示する。
F18-PF1-PF2	HELP KEYPAD MOVE_GOLD	キーパッド・キー図とその MOVE GOLD 機能を表示する。
F18-PF4-PF2	HELP KEYPAD MOVE_BLUE	キーパッド・キー図とその MOVE BLUE 機能を表示する。
F19-PF2	HELP KEYPAD EXPAND	キーパッド・キー図とその EXPAND DEFAULT 機能を表示する。
F19-PF1-PF2	HELP KEYPAD EXPAND_GOLD	キーパッド・キー図とその EXPAND GOLD 機能を表示する。
F19-PF4-PF2	HELP KEYPAD EXPAND_BLUE	キーパッド・キー図とその EXPAND BLUE 機能を表示する。
F20-PF2	HELP KEYPAD CONTRACT	キーパッド・キー図とその CONTRACT DEFAULT 機能を表示する。
F20-PF1-PF2	HELP KEYPAD CONTRACT_GOLD	キーパッド・キー図とその CONTRACT GOLD 機能を表示する。
F20-PF4-PF2	HELP KEYPAD CONTRACT_BLUE	キーパッド・キー図とその CONTRACT BLUE 機能を表示する。

A.5 デバッガのキー定義

表 A–4 に，すべてのキー定義を示します。

表 A–4 デバッガのキー定義

キー	状態	起動されるコマンドまたは機能
KP0	DEFAULT	STEP
	GOLD	STEP/INTO
	BLUE	STEP/OVER

(次ページに続く)

表 A–4 (続き) デバッガのキー定義

キー	状態	起動されるコマンドまたは機能
KP1	DEFAULT	EXAMINE。現在の要素の論理的後続データ (その次の記憶位置) が定義されている場合，それを検査する。
	GOLD	EXAMINE ^。現在の要素の論理的先行データ (その前の記憶位置) が定義されている場合，それを検査する。
	BLUE	定義済みのプロセス固有のソース・ディスプレイ SRC_ *n*と機械語命令ディスプレイ INST_*n*を 3 組表示する。すなわち，可視プロセスのソース・ディスプレイと機械語命令ディスプレイ (それぞれ S2 と RS 2 にある)，プロセス・リスト中の前行のプロセスのソース・ディスプレイと機械語命令ディスプレイ (それぞれ S1 と RS1 にある)，プロセス・リスト中の次行のプロセスのソース・ディスプレイと機械語命令ディスプレイ (それぞれ S3 と RS3 にある) を表示する。
KP2	DEFAULT	SCROLL/DOWN
	GOLD	SCROLL/BOTTOM
	BLUE	SCROLL/DOWN (コマンドは終了しない)。コマンド入力を終了させるには，スクロールする行数 (:*n*) またはディスプレイ名を指定する。
	MOVE	MOVE/DOWN
	MOVE_GOLD	MOVE/DOWN:999
	MOVE_BLUE	MOVE/DOWN:5
	EXPAND	EXPAND/DOWN
	EXPAND_GOLD	EXPAND/DOWN:999
	EXPAND_BLUE	EXPAND/DOWN:5
	CONTRACT	EXPAND/DOWN:-1
	CONTRACT_GOLD	EXPAND/DOWN:-999
	CONTRACT_BLUE	EXPAND/DOWN:-5
KP3	DEFAULT	SELECT/SCROLL %NEXTSCROLL。ディスプレイ・リスト中で現在のスクロール・ディスプレイの次のディスプレイを，現在のスクロール・ディスプレイとして選択する。
	GOLD	SELECT/OUTPUT %NEXTOUTPUT。ディスプレイ・リスト中の次の出力ディスプレイを現在の出力ディスプレイとして選択する。
	BLUE	3 つの定義済みプロセス固有のソース・ディスプレイ， SRC_*n*を表示する。それぞれ S1， S2， S3 にあり，プロセス・リストで現在の (可視) プロセスの前にあるプロセス，現在のプロセス，その次にあるプロセスのソース・ディスプレイである。
KP4	DEFAULT	SCROLL/LEFT
	GOLD	SCROLL/LEFT:255

(次ページに続く)

表 A–4 (続き) デバッガのキー定義

キー	状態	起動されるコマンドまたは機能
	BLUE	SCROLL/LEFT (非完結型)。コマンドを完結させるには，スクロールする行数 (:*n*) またはディスプレイ名を指定する。
	MOVE	MOVE/LEFT
	MOVE_GOLD	MOVE/LEFT:999
	MOVE_BLUE	MOVE/LEFT:10
	EXPAND	EXPAND/LEFT
	EXPAND_GOLD	EXPAND/LEFT:999
	EXPAND_BLUE	EXPAND/LEFT:10
	CONTRACT	EXPAND/LEFT:-1
	CONTRACT_GOLD	EXPAND/LEFT:-999
	CONTRACT_BLUE	EXPAND/LEFT:-10
KP5	DEFAULT	EXAMINE/SOURCE .%SOURCE_SCOPE\%PC; EXAMINE/INST .%INST_SCOPE\%PC。行 (非画面) モードで，次に実行するソース行と命令を表示する。画面モードで，現在のソース・ディスプレイを次に実行するソース行の中央におく。また，現在の機械語命令ディスプレイを次に実行する命令の中央におく。
	GOLD	SHOW CALLS
	BLUE	SHOW CALLS 3
KP6	DEFAULT	SCROLL/RIGHT
	GOLD	SCROLL/RIGHT:255
	BLUE	SCROLL/RIGHT (非完結型)。コマンドを完結させるには，スクロールする行数 (:*n*) またはディスプレイ名を指定する。
	MOVE	MOVE/RIGHT
	MOVE_GOLD	MOVE/RIGHT:999
	MOVE_BLUE	MOVE/RIGHT:10
	EXPAND	EXPAND/RIGHT
	EXPAND_GOLD	EXPAND/RIGHT:999
	EXPAND_BLUE	EXPAND/RIGHT:10
	CONTRACT	EXPAND/RIGHT:-1
	CONTRACT_GOLD	EXPAND/RIGHT:-999
	CONTRACT_BLUE	EXPAND/RIGHT:-10
KP7	DEFAULT	DISPLAY SRC AT LH1, INST AT RH1, OUT AT S45, PROMPT AT S6; SELECT/SCROLL/SOURCE SRC; SELECT/INST INST; SELECT/OUT OU T。ディスプレイ SRC， INST， OUT， PROMPT とその属性を表示する。

(次ページに続く)

表 A–4 (続き) デバッガのキー定義

キー	状態	起動されるコマンドまたは機能
	GOLD	DISPLAY INST AT LH1, REG AT RH1, OUT AT S45, PROMPT AT S6; SELECT/SCROLL/INST INST;SELECT/OUT OUT。ディスプレイ INST, REG, OUT, PROMPT に適切なディスプレイをつけて表示する。
	BLUE	定義済みのプロセス固有のソース・ディスプレイ SRC_ *n*と機械語命令ディスプレイ, INST_*n*を 2 組表示する。すなわち, 可視プロセスのソース・ディスプレイと機械語命令ディスプレイ (それぞれ Q1 と RQ1 にある), プロセス・リストの次行プロセスのソース・ディスプレイと機械語命令ディスプレイ (それぞれ Q2 と RQ2 にある) を表示する。
KP8	DEFAULT	SCROLL/UP
	GOLD	SCROLL/TOP
	BLUE	SCROLL/UP (非完結型)。コマンドを完結させるには, スクロールする行数 (:*n*) またはディスプレイ名を指定する。
	MOVE	MOVE/UP
	MOVE_GOLD	MOVE/UP:999
	MOVE_BLUE	MOVE/UP:5
	EXPAND	EXPAND/UP
	EXPAND_GOLD	EXPAND/UP:999
	EXPAND_BLUE	EXPAND/UP:5
	CONTRACT	EXPAND/UP:-1
	CONTRACT_GOLD	EXPAND/UP:-999
	CONTRACT_BLUE	EXPAND/UP:-5
KP9	DEFAULT	DISPLAY %NEXTDISP。ディスプレイ・リスト中の次のディスプレイを現在のウィンドウに表示する。ただし, 削除したディスプレイは除く。
	GOLD	SET PROCESS/VISIBLE %NEXT_PROCESS。プロセス・リスト中の次のプロセスを可視プロセスにする。
	BLUE	2 つの定義済みプロセス固有のソース・ディスプレイ SRC_ *n*を表示する。これらはそれぞれ Q1 (可視プロセス) と Q2 (プロセス・リスト中の次のプロセス) にある。
PF1		次に押すキーの GOLD 機能を有効にする。
PF2		キーパッド・キーのダイアグラムと DEFAULT 機能を表示する。
PF2		表 A–3 を参照。
PF3	DEFAULT	SET MODE SCREEN; SET STEP NOSOURCE。画面モードを使用可能にし, 通常は出力表示中に表示されるソース行を出力しないようにする (ソース・ディスプレイを行ったときにこの出力があると冗長であるため)。

(次ページに続く)

表 A–4 (続き) デバッガのキー定義

キー	状態	起動されるコマンドまたは機能
	GOLD	SET MODE NOSCREEN; SET STEP SOURCE。画面モードを禁止し，ソース行の出力を行えるように戻す。
	BLUE	DISPLAY/GENERATE。すべての自動更新ディスプレイの内容を再生成する。
PF4		次に押すキーの BLUE 機能を有効にする。
COMMA	DEFAULT	GO
	GOLD	SELECT/SOURCE %NEXT_SOURCE。ディスプレイ・リスト中の次のソース・ディスプレイを現在のソース・ディスプレイとして選択する。
	BLUE	SELECT/INSTRUCTION %NEXTINST。ディスプレイ・リスト中の次の機械語命令ディスプレイを現在の機械語命令ディスプレイとして選択する。
MINUS	DEFAULT	DISPLAY %NEXTDISP AT S12345, PROMPT AT S6; SELECT/SCROLL %CURDISP。ディスプレイ・リスト中の次のディスプレイを画面いっぱいに表示する (画面の最上部に PROMPT ディスプレイの最上部を合わせる)。そのディスプレイを現在のスクロール・ディスプレイとして選択する。
	GOLD	未定義
	BLUE	DISPLAY SRC AT H1, OUT AT S45, PROMPT AT S6; SELECT/SCROLL/SOURCE SRC; SELECT /OUT OUT。ディスプレイ SRC, OUT, PROMPT とその属性を表示する。これは省略時のディスプレイ構成である。
Enter		コマンドを入力する (完結させる)。 Return キーと同じである。
PERIOD	すべての状態	GOLD や BLUE などのようにその状態にロックをかけない状態キーを押した場合の効果を取り消す。 MOVE, EXPAND, CONTRACT などのように状態ロックをかける状態キーの操作には影響しない。
Next Screen (E6)	DEFAULT	SCROLL/DOWN
Prev Screen (E5)	DEFAULT	SCROLL/UP
Remove (E3)	DEFAULT	DISPLAY/REMOVE %CURSCROLL。ディスプレイ・リストから現在のスクロール・ディスプレイを削除する
Select (E4)	DEFAULT	SELECT/SCROLL %NEXTSCROLL。ディスプレイ・リスト中で現在のスクロール・ディスプレイの次のディスプレイを，現在のスクロール・ディスプレイとして選択する。
F17		キーパッドを DEFAULT 状態にし， KP2, KP4, KP6, KP8 のスクロール・ディスプレイ機能を有効にする。デバッガを起動すると，キーパッドは DEFAULT 状態になる。

(次ページに続く)

表 A–4 (続き) デバッガのキー定義

キー	状態	起動されるコマンドまたは機能
F18		キーパッドを MOVE 状態にし， KP2， KP4， KP6， KP8 の移動ディスプレイ機能を有効にする。
F19		キーパッドを EXPAND 状態にし， KP2， KP4， KP6， KP8 の拡大ディスプレイ機能を有効にする。
F20		キーパッドを CONTRACT 状態にし， KP2， KP4， KP6， KP8 の縮小ディスプレイ機能を有効にする。
Ctrl/W		DISPLAY/REFRESH
Ctrl/Z		EXIT

B

組み込みシンボルと論理名

この付録では，デバッガのすべての組み込みシンボルと論理名を示します。

B.1 SS$_DEBUG 条件

SS$_DEBUG (SYS$LIBRARY:STARLET.OLB で定義される) は，プログラムからデバッガを起動するようにシグナル通知する場合の条件です。プログラムから SS$_DEBUG をシグナル通知するのは， Ctrl/Y を押したあとに DCL コマンド DEBUG を入力することと同じです。

コマンドを SS$_DEBUG とともにシグナル通知して，デバッガにコマンドを渡すことができます。デバッガに実行させたいコマンドの形式は， DBG>プロンプトに対して入力するコマンドの形式と同じにしてください。複数のコマンドを指定する場合は，セミコロンで区切らなければなりません。また，コマンドは ASCIC 文字列として参照渡しで渡さなければなりません。 ASCIC 文字列を作成する方法についての詳しい説明は，各言語のドキュメントを参照してください。

たとえば，プログラムのある地点でデバッガを起動して SHOW CALLS コマンドを入力するには，次のコードをプログラムに挿入します (BLISS の例)。

```
LIB$SIGNAL(SS$_DEBUG, 1, UPLIT BYTE(%ASCIC 'SHOW CALLS'));
```

SS$_DEBUG の定義は，言語により STARLET ファイルまたは SYSDEF ファイル (たとえば， BLISS の STARLET.L32，または Fortran の FORSYSDEF.TLB) からコンパイルするときに得ることができます。また，リンク時に SYS$LIBRARY:STARLET.OLB から SS$_DEBUG の定義を得ることもできますが，この方法はなるべく使用しないでください。

B.2 論理名

デバッガ固有の論理名を次の表に示します。

論理名	機能
DBG$DECW$DISPLAY	DECwindows Motif for OpenVMS を実行しているワークステーションにだけ適用される。デバッガ・インタフェース (DECwindows Motif for OpenVMS またはコマンド) またはディスプレイ装置を指定する。省略時の設定: DBG$DECW$DISPLAY は未定義であるか，またはアプリケーション単位の論理名 DECW$DISPLAY 定義と同じ定義である。 DECwindows Motif for OpenVMS 環境でデバッガの省略時のインタフェースを変更するために，DBG$DECW$DISPLAY を使用する方法については，第 9.8.3 項を参照。
DBG$IMAGE_DSF_PATH	(Alpha および Integrity のみ) デバッグするイメージの.DSF (デバッグ・シンボル・テーブル) ファイルを格納するディレクトリを指定する。各.DSF ファイルのファイル名は，デバッグするイメージのファイル名と同じでなければならない。.DSF ファイルの作成の詳細については，第 5.1.5 項を参照。
DBG$INIT	デバッガの初期化ファイルを指定する。省略時の設定では，デバッガの初期化ファイルはない。 DBG$INIT は，検索リストだけでなく，ファイルの完全指定でも部分指定でもよい。デバッガの初期化ファイルについての詳しい説明は，第 13.2 節を参照。
DBG$INPUT	デバッガの入力装置を指定する。省略時の設定は SYS$INPUT である。 DBG$INPUT と DBG$OUTPUT を使用して 2 つの端末で画面用プログラムをデバッグする方法についての詳しい説明は，第 14.2 節を参照。 DBG$INPUT は DECwindows Motif for OpenVMS ユーザ・インタフェースでは無視される (DBG$DECW$DISPLAY を参照)。 DECterm ウィンドウでデバッガのコマンド・インタフェースを表示中であれば，DBG$INPUT を使用できる。
DBG$OUTPUT	デバッガの出力装置を指定する。省略時の設定は SYS$OUTPUT である。 DBG$INPUT と DBG$OUTPUT を使用して 2 つの端末で画面用プログラムをデバッグする方法については，第 14.2 節を参照。 DECwindows Motif for OpenVMS ユーザ・インタフェースでは DBG$OUTPUT は無視される (DBG$DECW$DISPLAY を参照)。 DECterm ウィンドウでデバッガのコマンド・インタフェースを表示中であれば， DBG$OUTPUT を使用できる。
SSI$AUTO_ACTIVATE	(Alpha のみ) システム・サービス・インターセプション (SSI) が有効であるかどうか指定する。ウォッチポイントで問題がある場合は，次の DCL コマンドを入力し， SSI を無効にする。 `$DEFINE SSI$AUTO_ACTIVATE OFF`静的ウォッチポイント，AST，システム・サービス・インターセプションの相互関係については， SET WATCH の説明 を参照。

論理名に値を割り当てるには， DCL コマンド DEFINE または ASSIGN を使用します。たとえば，次のコマンドは，デバッガの初期化ファイルの記憶位置を指定します。

```
$ DEFINE DBG$INIT DISK$:[JONES.COMFILES]DEBUGINIT.COM
```

論理名 DBG$INPUT については，次の点に注意してください。ファイル (たとえば PROG_IN.DAT) から入力を得るプログラムおよび，端末からデバッガ入力を得るプログラムをデバッグする場合は，次のように定義してから，デバッガを起動しなければなりません。

```
$ DEFINE SYS$INPUT PROG_IN.DAT
$ DEFINE/PROCESS DBG$INPUT 'F$LOGICAL("SYS$COMMAND")
```

すなわち， SYS$COMMAND の変換を指すように DBG$INPUT を定義します。SYS$COMMAND を指すように DBG$INPUT を定義すると，デバッガは PROG_IN.DAT ファイルから入力を得ようとします。

B.3 組み込みシンボル

デバッガの組み込みシンボルは，プログラムの要素と値を指定するオプションです。

デバッガのほとんどの組み込みシンボルには，パーセント記号接頭辞(%)があります。

この付録では，次のシンボルについて説明します。

- %NAME— 識別子を作成する。
- %PARCNT— コマンド・プロシージャ内で使用し，渡すパラメータを数える。
- %DECWINDOWS— デバッガ・コマンド・プロシージャ内または初期化ファイル内で使用し，デバッガのコマンド・インタフェースと DECwindows Motif for OpenVMS ユーザ・インタフェースのどちらが表示されるかを示す。
- %BIN, %DEC, %HEX, %OCT— 入力基数を制御する。
- ピリオド(.)， Return キー，サーカンフレックス(^)，バックスラッシュ(\ )，%CURLOC，%NEXTLOC，%PREVLOC，%CURVAL— 連続したプログラム記憶位置および要素の現在の値を指定する。
- + 記号(+)， - 記号(-)，乗算記号(*)，除算記号(/)，アットマーク(@)，ピリオド(.)，ビット・フィールド演算子 (<p,s,e>)，%LABEL，%LINE，バックスラッシュ(\ )— アドレス式の演算子として使用する。
- %ADAEXC_NAME，%EXC_FACILITY，%EXC_NAME，%EXC_NUMBER，%EXC_SEVERITY— 例外についての情報を入手するのに使用する。
- %CURRENT_SCOPE_ENTRY，%NEXT_SCOPE_ENTRY，%PREVIOUS_SCOPE_ENTRY— 呼び出しスタックに対して現在の有効範囲，次の有効範囲，前の有効範囲を指定する。
- Alpha プロセッサの場合
  - %R0 ～%R28，%FP，%SP，%R31，%PC，%PS — Alpha 汎用レジスタを指定する。

    - %F0 ～%F30，%F31 — Alpha 浮動小数点レジスタを指定する。
- Integrity プロセッサの場合
    - %R0 ～%R127，%GP，%SP，%TP，%AP，%OUT0 ～%OUT7 — Integrity 汎用レジスタを指定する。
    - %F0 ～%F127 — Integrity 浮動小数点レジスタを指定する。
    - P0 ～%P63，%PR — Integrity プレディケート・レジスタを指定する。
    - %B0 ～%B7，%RP — Integrity 分岐レジスタを指定する。
    - %AR0 ～%AR7，%AR17 ～%AR19，%AR32，%AR36，%AR40，%AR64 ～%AR66，%KR0 ～%KR7，%RSC，%BSP，%RNAT，%CCV，%UNAT，%FPSR，%PFS，%L — Integrity アプリケーション・レジスタを指定する。
    - %CR0 ～%CR2，%CR8，%CR16，%CR17，%CR19 ～%CR25，%CR64，%CR66，%CR68 ～%CR74，%CR80，%CR81，%DCR，%ITM，%IVA，%PTA，%PSR，%IPSR，%ISR，%IIP, %IFA，%ITIR，%IIPA，%IFS，%IIM，%IHA，%LID，%TPR，%IRR0 ～%IRR3，%ITV，%PMV，%CMCV，%IRR0 ～%IRR3 — Integrity コントロール・レジスタを指定する。
    - %SR0，%IH，%PREV_BSP，%PC，%IP，%RETURN_ PC，%CFM，%NEXT_PFS，%PSP，%CHTCTX_ ADDR，%OSSD，%HANDLER_FV，%LSDA，%UM — Integrity 特殊レジスタを指定する。
- %ADDR，%DESCR，%REF，%VAL—CALL コマンドの引数受け渡し方式を指定する。コマンド・ディクショナリの CALL コマンドの説明を参照。
- %PROCESS_NAME，%PROCESS_PID，%PROCESS_NUMBER，%NEXT_ PROCESS，%PREVIOUS_PROCESS，%VISIBLE_PROCESS— マルチプロセス・プログラムで使用するプロセスを指定する。第 15.16.1 項を参照。
- %ACTIVE_TASK，%CALLER_TASK，%NEXT_TASK，%PREVIOUS_ TASK，%TASK，%VISIBLE_TASK— タスキング・プログラムまたはマルチスレッド・プログラムのタスクまたはスレッドを指定する。第 16.3.4 項を参照。
- %PAGE，%WIDTH— 現在の画面の高さと幅を指定する。第 7.11.1 項を参照。
- %SOURCE_SCOPE，%INST_SCOPE— 画面モードでソース表示と機械語命令ディスプレイの有効範囲を指定する。それぞれ第 7.4.1 項と第 7.4.4 項を参照。
- %CURDISP，%CURSCROLL，%NEXTDISP，%NEXTINST, %NEXTOUTPUT，%NEXTSCROLL，%NEXTSOURCE— 表示リスト中の表示を指定するため画面モードで使用する。第 7.11.2 項を参照。

B.3.1 レジスタの指定

VAX レジスタ， Alpha レジスタ，または Integrity レジスタのデバッガ組み込みシンボルは，レジスタ名の前にパーセント記号(%)がついたものです。レジスタ・シンボルを指定する場合，同じ名前のシンボルをプログラムが宣言していなければパーセント記号(%)を前につける必要はありません。

すべてのレジスタの内容を検査できます。また， SP を除くすべてのレジスタに値を格納することもできます。ただし， FP に値を格納するときは注意してください。

表 B–1 に Alpha レジスタ・シンボルを示します。

表 B–1 Alpha レジスタのデバッガ・シンボル (Alpha のみ)

シンボル	機能
	Alpha 整数レジスタ
%R0 . . . %R28	レジスタ R0 . . . R28
%FP (%R29)	スタック・フレーム基底レジスタ (FP)
%SP (%R30)	スタック・ポインタ (SP)
%R31	ReadAsZero/Sink (RZ)
%PC	プログラム・カウンタ (PC)
%PS	プロセッサ・ステータス・レジスタ (PS)。組み込みシンボル%PSL と%PSW は Alpha プロセッサでは使用できない。
	Alpha 浮動小数点レジスタ
%F0 . . . %F30	レジスタ F0 . . . F30
%F31	ReadAsZero/Sink

デバッガには，画面モードのレジスタ表示は用意されていません。

Alpha プロセッサの場合

- R31 レジスタや F31 レジスタには値を格納できない。この 2 つのレジスタには，永久的に値 0 が割り当てられている。
- ベクタ・レジスタがない。

Alpha 汎用レジスタについての詳しい説明は，第 4.4 節と第 4.4.1 項を参照してください。

表 B–2 に， Integrity レジスタのシンボルを示します。

表 B–2 Integrity レジスタのデバッガ・シンボル (Integrity のみ)

シンボル	説明
	Integrity アプリケーション・レジスタ
%KR0 . . . %KR7	カーネル・レジスタ 0 . . . 7
%RSC (%AR16)	レジスタ・スタック・コンフィギュレーション
%BSP (%AR17)	バッキング・ストア・ポインタ
%BSPSTORE (%AR18)	メモリ・ストア用バッキング・ストア・ポインタ
%RNAT (%AR19)	RSE NaT コレクション
%CCV ($AR32)	比較交換での比較値
%UNAT (%AR36)	ユーザ NaT コレクション
%FPSR (%AR40)	浮動小数点ステータス
%PFS (%AR64)	以前のファンクション状態
%LC (%AR65)	ループ・カウント
%EC (%AR66)	エピローグ・カウント
%CSD	コード・セグメント
%SSD	スタック・セグメント

(次ページに続く)

表 B–2 (続き) Integrity レジスタのデバッガ・シンボル (Integrity のみ)

シンボル	説明
	コントロール・レジスタ
%DCR (%CR0)	デフォルト・コントロール
%ITM (%CR1)	インターバル・タイマ・マッチ (SCD でのみ参照可能)
%IVA (%CR2)	割り込みベクタ・アドレス (SCD でのみ参照可能)
%PTA (%CR8)	ページ・テーブル・アドレス (SCD でのみ参照可能)
%PSR (%CR9, %ISPR)	割り込みプロセッサ・ステータス
%ISR (%CR17)	割り込みステータス
%IIP (%CR19)	割り込み命令ポインタ
%IFA (%CR20)	割り込みフォルト・アドレス
%ITIR (%CR21)	割り込み TLB 挿入
%IIPA (%CR22)	割り込み命令前アドレス
%IFS (%CR23)	割り込みファンクション状態
%IIM (%CR24)	割り込み即値
%IHA (%CR25)	割り込みハッシュ・アドレス
%LID (%CR64)	ローカル割り込み ID (SCD でのみ参照可能)
%TPR (%CR66)	タスク・プライオリティ (SCD でのみ参照可能)
%IRR0 . . . %IRR3 (%CR68 . . . %CR71)	外部割り込み要求 0 . . . 3 (SCD でのみ参照可能)
%ITV (%CR72)	インターバル・タイマ (SCD でのみ参照可能)
%PMV (%CR73)	パフォーマンス監視 (SCD でのみ参照可能)
%CMCV (%CR74)	訂正済みマシン・チェック・ベクタ (SCD でのみ参照可能)
%IRR0 および%IRR1 (%CR80 および%CR81)	ローカル・リダイレクション 0:1 (SCD でのみ参照可能)

(次ページに続く)

表 B–2 (続き) Integrity レジスタのデバッガ・シンボル (Integrity のみ)

シンボル	説明
	特殊レジスタ
%IH (%SR0)	インボケーション・ハンドル
%PREV_BSP	以前のバッキング・ストア・ポインタ
%PC (%IP)	プログラム・カウンタ (命令ポインタ | スロット番号)
%RETURN_PC	リターン・プログラム・カウンタ
%CFM	現在のフレーム・マーカ
%NEXT_PFS	前々回のファンクション状態
%PSP	以前のスタック・ポインタ
%CHFCTX_ADDR	コンディション・ハンドリング・ファシリティ・コンテキスト・アドレス
%OSSD	オペレーティング・システム固有データ
%HANDLER_FV	ハンドラ・ファンクション値
%LSDA	言語固有データ領域
%UM	ユーザ・マスク
	プレディケート・レジスタ
%PR (%PRED)	プレディケート・コレクション・レジスタ —%P0 . . . %P63 の集まり
%P0 . . . %P63	プレディケート (1 ビット) レジスタ 0 . . . 63
	分岐レジスタ
%RP (%B0)	リターン・ポインタ
%B1 . . . %B7	分岐レジスタ 1 . . . 7
	汎用整数レジスタ
%R0	汎用整数レジスタ 0
%GP (%R1)	グローバル・データ・ポインタ
%R2 . . . %R11	汎用整数レジスタ 2 . . . 11
%SP (%R12)	スタック・ポインタ
%TP (%R13)	スレッド・ポインタ
%R14 . . . %R24	汎用整数レジスタ 14 . . . 24
%AP (%R25)	引数情報
%R26 . . . %R127	汎用整数レジスタ 26 . . . 127
	出力レジスタ
%OUT0 . . . %OUT7	出力レジスタ，実行時別名 (たとえば，フレームが出力レジスタに割り当てられた場合，%OUT0 は，最初に割り当てられた出力レジスタ，たとえば%R38 に対応する)。

(次ページに続く)

表 B–2 (続き) Integrity レジスタのデバッガ・シンボル (Integrity のみ)

シンボル	説明
	汎用レジスタ
%GRNAT0 および%GRNAT1	それぞれ 64 ビットの汎用レジスタの NAT (Not A Thing) コレクション・レジスタ。たとえば%GRNAT0<3,1,0>は，%R3 の NAT ビット。
	浮動小数点レジスタ
%F0 . . . %F127	浮動小数点レジスタ 0 . . . 127

Integrity レジスタについての詳細は，第 4.4 節および第 4.4.2 項を参照してください。

B.3.2 識別子の作成

組み込みシンボル%NAME を使用すると，現在の言語では通常は使用できない識別子を作成できます。この場合の構文は次のとおりです。

```
%NAME 'character-string'
```

次の例では，'12' という名前の変数を調べます。

```
DBG> EXAMINE %NAME '12'
```

次の例では，コンパイラ生成ラベル P.AAA を調べます。

```
DBG> EXAMINE %NAME 'P.AAA'
```

B.3.3 コマンド・プロシージャに渡されるパラメータの数

組み込みシンボル%PARCNT は，指定する実パラメータの個数が一定でないコマンド・プロシージャで使用します。%PARCNT はデバッガ・コマンド・プロシージャ内だけで定義されます。

%PARCNT は，現在のコマンド・プロシージャに渡される実パラメータの数を指定します。次の例では，コマンド・プロシージャ ABC.COM が起動され， 3 つのパラメータが渡されます。

```
DBG> @ABC 111,222,333
```

ABC.COM では，%PARCNT の値は現在 3 です。この%PARCNT は ABC.COM に渡された各パラメータ値を得るためのループ・カウンタとして使用されます。

```
DBG> FOR I = 1 TO %PARCNT DO (DECLARE X:VALUE; EVALUATE X)
```

B.3.4 デバッガ・インタフェース (コマンドまたは DECwindows Motif for OpenVMS) の決定

組み込みシンボル%DECWINDOWS を使用すると，デバッガの DECwindows Motif for OpenVMS インタフェースが表示されるのかコマンド・インタフェースが表示されるのかを決定できます。 DECwindows Motif for OpenVMS ユーザ・インタフェースを使用していれば，%DECWINDOWS の値は 1 (TRUE) です。コマンド・インタフェースを使用していれば，%DECWINDOWS の値は 0 (FALSE) です。次に例を示します。

```
DBG> EVALUATE %DECWINDOWS
0
```

次の例は，デバッガの初期化ファイル内で%DECWINDOWS を使用して，デバッガの起動時にデバッガ・ソース・ウィンドウ SRC の位置決めする方法を示します。

```
IF %DECWINDOWS THEN
      ! DECwindows Motif (ワークステーション) 構文:
      (DISPLAY SRC AT (100,300,100,700))
   ELSE
      ! 画面モード (端末) 構文:
      (DISPLAY SRC AT (AT H1))
```

B.3.5 入力基数の制御

組み込みシンボル%BIN，%DEC，%HEX，%OCT は，アドレス式と言語式内で，そのあとに続く整数リテラル，またはそのあとの括弧で囲まれた式の中にあるすべての整数リテラルをそれぞれ， 2 進基数， 10 進基数， 16 進基数， 8 進基数と解釈することを指定するのに使用します。ただし，これらの基数組み込みシンボルは，必ず整数リテラルとともに使用しなければなりません。次に例を示します。

```
DBG> EVALUATE/DEC %HEX 10
16
DBG> EVALUATE/DEC %HEX (10 + 10)
32
DBG> EVALUATE/DEC %BIN 10
2
DBG> EVALUATE/DEC %OCT (10 + 10)
16
DBG> EVALUATE/HEX %DEC 10
0A
DBG> SET RADIX DECIMAL
DBG> EVALUATE %HEX 20 + 33 ! 20 を 16 進数として，33 を 10 進数として扱う
65                         ! 結果として得られる値は 10 進数である
DBG> EVALUATE %HEX (20+33) ! 20 と 33 の両方を 16 進数として扱う
83
DBG> EVALUATE %HEX (20+ %OCT 10 +33) ! 20 と 33 を 16 進数として，
91                                   ! 10 を 8 進数として扱う
DBG> SYMBOLIZE %HEX 27C9E3  ! 16 進アドレスをシンボル化する
DBG> DEPOSIT/INST %HEX 5432 = 'MOVL ^O%DEC 222, R1'
DBG> ! アドレス 5432 を 16 進数として，オペランド 222 を10 進数として扱う
```

B.3.6 プログラム記憶位置と要素の現在値の指定

次の組み込みシンボルを使用すると，プログラム記憶位置と要素の現在値を指定できます。

シンボル	機能
%CURLOC .(ピリオド)	現在の論理要素 —EXAMINE コマンド， DEPOSIT コマンドまたは EVALUATE/ADDRESS コマンドによって最後に参照されたプログラム記憶位置。
%NEXTLOC Return キー	現在の要素の論理的後続データ —EXAMINE コマンド，DEPOSIT コマンド，または EVALUATE/ADDRESS コマンドによって最後に参照された記憶位置の論理的に続くプログラム記憶位置。 Return キーはコマンド終了文字なので，これはコマンド終了文字を指定できる場所 (たとえば EXAMINE の直後，ただし DEPOSIT または EVALUATE/ADDRESS の直後ではいけない) だけで使用できる。
%PREVLOC ^(サーカンフレックス)	現在の要素の論理的先行データ —EXAMINE コマンド，DEPOSIT コマンドまたは EVALUATE/ADDRESS コマンドによって最後に参照された記憶位置の論理的に前にあるプログラム記憶位置。
%CURVAL \ (バックスラッシュ)	EVALUATE コマンドまたは EXAMINE コマンドによって最後に表示されたか DEPOSIT コマンドによって最後に格納された値。この 2 つのシンボルは， EVALUATE/ADDRESS コマンドの影響を受けない。

次の例では， WIDTH 変数を調べます。次に，値 12 を，現在の記憶位置 (WIDTH) に格納します。これは，現在の記憶位置を確認することによって調べます。

```
DBG> EXAMINE WIDTH
MOD\WIDTH: 7
DBG> DEPOSIT . = 12
DBG> EXAMINE .
MOD\WIDTH: 12
DBG> EXAMINE %CURLOC
MOD\WIDTH: 12
DBG>
```

次の例では，配列内で次の記憶位置と前にある記憶位置を調べます。

```
DBG> EXAMINE PRIMES(4)
MOD\PRIMES(4): 7
DBG> EXAMINE %NEXTLOC
MOD\PRIMES(5): 11
DBG> EXAMINE Return     ! 次の記憶位置を調べる
MOD\PRIMES(6): 13
DBG> EXAMINE %PREVLOC
MOD\PRIMES(5): 11
DBG> EXAMINE ^
MOD\PRIMES(4): 7
DBG>
```

すべての場合に， Return キーを使用して論理的後続データを示すことができるとはかぎりません。たとえば，次の記憶位置を示すにはシンボル%NEXTLOC を使用しますが，その目的で DEPOSIT コマンドの入力後に Return キーを押すことはできません。

B.3.7 アドレス式におけるシンボルと演算子の使用方法

アドレス式で使用できるシンボルと演算子を次に示します。単項演算子のオペランドは 1 つです。二項演算子のオペランドは 2 つです。

シンボル	機能
%LABEL	後続の数値リテラルがプログラム・ラベル (Fortran などのように数値のプログラム・ラベルを持っている言語の場合) であることを指定する。含んでいるモジュールを指定するパス名接頭識別子をラベルにつけることができる。
%LINE	後続の数値リテラルがプログラム内の行番号であることを指定する。含んでいるモジュールを指定するパス名接頭識別子を行番号につけることができる。
バックスラッシュ(\)	パス名内で使用すると，パス名の各要素を区切る。この場合，完全パス名の一番左の要素がバックスラッシュであってはならない。 シンボルの接頭辞として使用する場合，シンボルをグローバル・シンボルと解釈することを指定する。この場合，バックスラッシュはシンボルの完全パス名の一番左の要素でなければならない。

シンボル	機能
アットマーク(@) ピリオド(.)	単項演算子。アドレス式では，アットマーク(@)とピリオド(.)はどちらも"内容"演算子として機能する。"内容"演算子を使用すると，そのオペランドはメモリ・アドレスと解釈され，そのアドレスの内容 (そこにある値) を指す。
ビット・フィールド<p,s,e>	単項演算子。ビット・フィールド選択をアドレス式に適用できる。ビット・フィールドを選択するには，ビット・オフセット(p)，ビット長(s)，符号拡張ビット(e)を指定する (省略可能)。
+ 記号(+)	単項演算子または二項演算子。単項演算子の場合，オペランドの値そのものを示す。二項演算子の場合，先行オペランドと後続オペランドの両方を加算する。
- 記号(-)	単項演算子または二項演算子。単項演算子の場合，オペランドの値の否定を示す。二項演算子の場合，先行オペランドから後続オペランドを減算する。
乗算記号(*)	二項演算子。先行オペランドに後続オペランドを掛ける。
除算記号(/)	二項演算子。先行オペランドを後続オペランドで割る。

次に，アドレス式での組み込みシンボルと演算子の使い方の例を示します。

%LINE 演算子と%LABEL 演算子

次のコマンドは，現在実行が中断しているモジュールの行 26 にトレースポイントを設定します。

```
DBG> SET TRACE %LINE 26
```

次のコマンドは，行 47 のソース行を表示します。

```
DBG> EXAMINE/SOURCE %LINE 47
module MAIN
    47:  procedure SWAP(X,Y: in out INTEGER) is
DBG>
```

次のコマンドは， MOD4 モジュールのラベル 10 にブレークポイントを設定します。

```
DBG> SET BREAK MOD4\%LABEL 10
```

パス名演算子

次のコマンドは， MOD4 モジュールの ROUT2 ルーチンで宣言された COUNT 変数の値を表示します。バックスラッシュ(\)パス名区切り文字でパス名要素を区切ります。

```
DBG> EXAMINE MOD4\ROUT2\COUNT
MOD4\ROUT2\COUNT: 12
DBG>
```

次のコマンドは， QUEUMAN モジュールの行 26 にブレークポイントを設定します。

```
DBG> SET BREAK QUEUMAN\%LINE 26
```

次のコマンドはグローバル・シンボル X の値を表示します。

```
DBG> EXAMINE \X
```

算術演算子

デバッガがアドレス式の要素を評価する順番は，ほとんどのプログラミング言語で使用される順番と似ています。この順番は，次の 3 つの要素によって決まります。優先順位が最も高いものから順に並んでいます。

1. 演算子ごとにオペランドをまとめるために使用する区切り文字。通常は括弧または大括弧。
2. 各演算子の相対優先順位。
3. 演算子間の優先順位。左側の演算子ほど高い。

次にデバッガ演算子を，優先順位が最も高いものから順に示します。

1. 単項演算子 (.，@， +， –)
2. 乗算演算子と除算演算子 (*，/)
3. 加算演算子と減算演算子 (+， –)

たとえば，次の式を評価する場合，デバッガはまず括弧内のオペランドを加算し，次にその結果を 4 で割り，次に 5 からその結果を引きます。

```
5--(T+5)/4
```

次のコマンドは，メモリ記憶位置 X + 4 バイトに含まれている値を表示します。
X + 4 bytes:

```
DBG> EXAMINE X + 4
```

内容演算子

次に，内容演算子の使い方を示します。最初の例は，現在の PC 値が指す命令 (アドレスが PC に含まれており，実行されようとしている命令) が得られます。

```
DBG> EXAMINE .%PC
MOD\%LINE 5: PUSHL   S^#8
DBG>
```

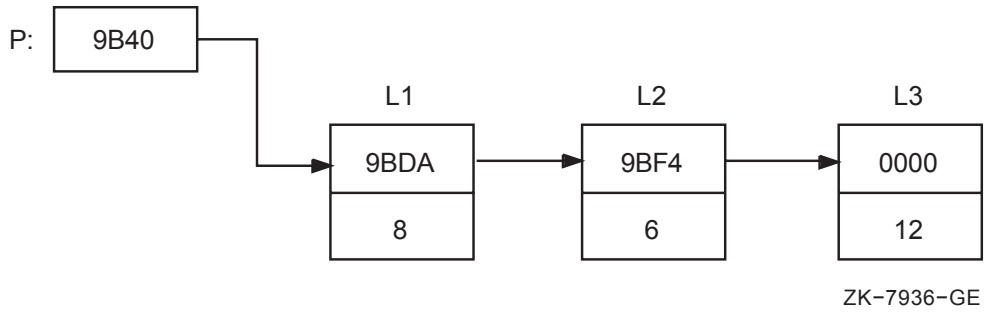
次の例では，呼び出しスタックの 1 レベル下にある PC 値が指すソース行が得られます (SWAP ルーチンを呼び出したとき)。

```
DBG> EXAMINE/SOURCE .1\%PC
module MAIN
MAIN\%LINE 134:     SWAP(X,Y);
DBG>
```

次の例では，ポインタ変数 PTR の値が 16 進数 7FF00000 であり，検査しようとしている要素のアドレスです。また，この要素の値は 16 進数の 3FF00000 です。次のコマンドは，この要素を検査する方法を示します。

```
DBG> EXAMINE/LONG .PTR
7FF00000: 3FF00000
DBG>
```

次の例では，内容演算子 (アットマークまたはピリオド) と現在の記憶位置演算子 (ピリオド) を使用して，3 つのクォドワード整数ポインタ変数がリンクされたリスト (次の図の L1，L2，L3) を検査します。P はリストの開始点へのポインタです。各ポインタ変数の上の方のロングワードには次の変数のアドレスが入り，下の方のロングワードには整数値 (それぞれ 8，6，12) が入っています。




```
DBG> SET TYPE QUADWORD; SET RADIX HEX
DBG> EXAMINE .P
! アドレスが P に入っている要素を
                              ! 検査する。
00009BC2: 00000008 00009BDA  ! 高位のワードには値 8 が，低位のワードには
                                 ! 次の要素のアドレス (9BDA) が入っている。

DBG> EXAMINE @.              ! アドレスが現在の
                             ! 要素に入っている要素を検査する。
00009BDA: 00000006 00009BF4  ! 高位のワードには値 6 が，低位のワードには
                                 ! 次の要素のアドレス (9BF4) が入っている。

DBG> EXAMINE ..              ! アドレスが現在の
                             ! 要素に入っている要素を検査する。
00009BF4: 0000000C 00000000  ! 高位のワードには値 12 (10 進数) が，低位の
                             ! ワードにはアドレス 0(リストの終わり) が入っている。
```

ビット・フィールド演算子

次の例は，ビット・フィールド演算子の使い方を示しています。たとえば，ビット 3 で始まり，長さが 4 ビット，符号拡張子がないアドレス式 X_NAME を検査するには，次のコマンドを入力します。

```
DBG> EXAMINE X_NAME <3,4,0>
```

B.3.8 例外情報の入手

次の組み込みシンボルを使用すると，現在の例外についての情報を入手したり，その情報を使用してブレークポイントまたはトレースポイントを修飾したりできます。

シンボル	機能
%EXC_FACILITY	現在の例外を発生させたファシリティの名前
%EXC_NAME	現在の例外の名前
%ADAEXC_NAME	現在の例外の Ada 例外名 (Ada プログラム専用)
%EXC_NUMBER	現在の例外の番号
%EXC_SEVERITY	現在の例外の重大度コード

次に例を示します。

```
DBG> EVALUATE %EXC_NAME
"FLTDIV_F"
DBG> SET BREAK/EXCEPTION WHEN (%EXC_NAME = "FLTDIV_F")
   .
   .
   .
DBG> EVALUATE %EXC_NUMBER
12
DBG> EVALUATE/CONDITION_VALUE %EXC_NUMBER
%SYSTEM-F-ACCVIO, access violation at PC !XL, virtual address !XL
DBG> SET BREAK/EXCEPTION WHEN (%EXC_NUMBER = 12)
```

WHEN 句の条件式は，言語によって異なります。

B.3.9 呼び出しスタック上の現在の有効範囲，次の有効範囲，前の有効範囲の指定

次の組み込みシンボルを使用すると，シンボル検索の有効範囲およびルーチン呼び出しスタックに関するソース表示と機械語命令ディスプレイの有効範囲を入手し，操作することができます。

組み込みシンボル	機能
%CURRENT_SCOPE_ENTRY	ソース・コードまたはデコードした命令を表示するときか，シンボルを検索するときに，デバッガが参照用に現在使用している呼び出しフレーム。省略時の設定は，呼び出しフレーム 0 である。
%NEXT_SCOPE_ENTRY	呼び出しスタックで，%CURRENT_SCOPE_ENTRY が示す呼び出しフレームの下にある次の呼び出しフレーム。
%PREVIOUS_SCOPE_ENTRY	呼び出しスタックで，%CURRENT_SCOPE_ENTRY が示す呼び出しフレームの上にある次の呼び出しフレーム。

これらのシンボルは，呼び出しスタック上の呼び出しフレームを示す整数値を返します。呼び出しフレーム 0 は，スタックの一番上にあるルーチンを示します。ここで実行が中断されています。呼び出しフレーム 1 は呼び出しルーチンを示します。

たとえば，次のコマンドは，デバッガが呼び出しスタックの 1 つ下のルーチンが示す有効範囲で始まるシンボル (呼び出しスタックの一番上にあるルーチンから始まるのではない) を検索することを指定します。

```
DBG> SET SCOPE/CURRENT %NEXT_SCOPE_ENTRY
```

C

各言語に対するデバッガ・サポートの要約

OpenVMS Debugger は， Integrity サーバおよび Alpha システムでさまざまなプログラミング言語をサポートします。

Integrity サーバでは，以下の言語で作成されたプログラムをデバッガで扱うことができます。

Ada[1]	Assembler (IAS)	BASIC	BLISS
C	C++	COBOL	Fortran
MACRO–32[2]	IMACRO	Pascal	

[1]Integrity サーバは AdaCore の GNAT Pro Ada 95 コンパイラをサポートします。

[2]MACRO–32 は AMACRO コンパイラでコンパイルされていなければなりません。

Alpha システムでは，以下の言語で作成されたプログラムをデバッガで扱うことができます。

HPE–Ada	BASIC	BLISS	C
C++	COBOL	Fortran	MACRO–32[1]
MACRO–64	Pascal	PL/I	

[1]MACRO–32 は AMACRO コンパイラでコンパイルされていなければなりません。

C.1 概要

デバッガは，各言語の構文，データ型，有効範囲規則を認識します。また，各言語の演算子および式の構文も認識します。したがって，デバッガ・コマンドを使用するときは，変数やその他のプログラム要素をプログラムのソース・コードに記述する場合と同じように指定することができます。また，その言語の構文を使用してソース言語の式の値を計算することもできます。

サポートされている言語のほとんどに共通するデバッグ方法については，マニュアルで説明しています。ヘルプ・トピックには各言語に固有の次の内容が含まれています。

- サポートされている言語式の演算子
- サポートされている言語式とアドレス式の構造
- サポートされているデータ型

- デバッガのサポートに制約がある場合はそれも含めた，各言語固有のその他の情報

言語固有のデバッガのサポートについての詳しい説明は，各言語とともに提供されるドキュメントを参照してください。

プログラムが複数の言語で記述されている場合，デバッグ・セッションの途中でデバッグ・コンテキストを 1 つの言語から別の言語へ変更することができます。選択する言語に対応したキーワードを指定して， SET LANGUAGE コマンドを使用してください。

Integrity サーバでは，次のキーワードの 1 つを指定することができます。

AMACRO	BASIC	BLISS	C
C++	COBOL	Fortran	PASCAL
UNKNOWN			

Alpha システムでは，次のキーワードの 1 つを指定することができます。

ADA	AMACRO	BASIC	BLISS
C	C++	COBOL	FORTRAN
MACRO	MACRO64	PASCAL	UNKNOWN

サポートされていない言語で記述されたプログラムをデバッグするときは， SET LANGUAGE UNKNOWN コマンドを入力します。サポートされていない言語でのデバッガの使用性を最大限に高めるため，この設定によってデバッガはデータ形式と演算子のセットを幅広く受け入れるようになります。この中には，サポートされている言語のうち， 2， 3 の言語だけに固有のデータ形式と演算子も含まれます。言語が UNKNOWN に設定されているときにデバッガが認識する演算子と構造についての詳しい説明は，オンライン・ヘルプのLanguage_UNKNOWNを参照してください。

C.2 GNAT Ada (Integrity のみ)

OpenVMS Integrity では， GNAT Pro (Ada 95) コンパイラがサポートされます。この製品の詳細については， Adacore 社までお問い合せください。

注意

HPE は，OpenVMS Alpha から OpenVMS Integrity へ HPE Ada (Ada 83) コンパイラをポーティングしていません。

Integrity サーバでは， AdaCore Technologies, Inc. の GNAT Pro Ada 95 を使用します。この製品に関する情報は， AdaCore 社の以下のドキュメントを参照してください。

- 『GNAT Pro Users Guide』 – このドキュメントでは， Ada 95 プログラミング言語のためのコンパイラおよびソフトウェア開発ツールセットである GNAT Pro の使用方法について説明しています。以下の URL で参照できます。

  http://www.gnat.com/wp-content/files/auto_update/gnat-unw-docs/html/gnat_ugn.html

- 『GNAT Pro Reference Manual』 — このドキュメントには， GNAT Pro コンパイラを使用してプログラムを作成するための情報が含まれています。このドキュメントでは， Annex M の標準で必要とされるすべての情報を含め， GNAT Pro の実装に依存した属性についても説明してます。以下の URL で参照できます。

  http://www.gnat.com/wp-content/files/auto_update/gnat-unw-docs/html/gnat_rm.html

OpenVMS Alpha 用の HPE Ada については，第 C.3 節を参照してください。

C.3 HPE Ada (Alpha)

次の各サブトピックでは， Alpha システムにおける HPE Ada のデバッガ・サポートについて説明します。 Ada のタスキング・プログラムに固有の情報については，第 16 章も参照してください。

C.3.1 Ada **の名前とシンボル**

次の各サブトピックでは，デバッガがサポートしている Ada の名前，シンボル，および定義済みの属性について説明します。

名前の一部分は言語式 (たとえば， 'FIRST や 'POS といった属性) である場合があるので注意してください。このことは， EXAMINE， EVALUATE， DEPOSIT の各コマンドでこれらの名前を使用する方法に影響します。列挙型の例については，オンライン・ヘルプの Specifying_Attributes_with_Enumeration_Types を参照してください。

C.3.1.1 Ada **の名前**

サポートされている Ada の名前を次に示します。

名前の種類	デバッガのサポート
レキシカル要素	Ada の識別子の構文規則を完全にサポートしている。
	識別子ではなく演算子のシンボル (たとえば，+ や*) である関数の名前には，接頭辞%NAME を付ける必要がある。また，演算子のシンボルは二重引用符で囲まなければならない。
	数値リテラル，文字リテラル，文字列リテラル，予約語についての Ada の規則を完全にサポートしている。
	-2147483648 から 2147483647 までの範囲で，符号付き整数リテラルを受け入れる。
	コンテキストとアーキテクチュアに応じて浮動小数点数型を，F 浮動小数点数型， D 浮動小数点数型， G 浮動小数点数型， H 浮動小数点数型， S 浮動小数点型，または T 浮動小数点数型と解釈する。
添字付き要素	完全にサポートしている。
断面	断面全体，または断面の添字付き要素を確認および評価することができる。
	断面の添字付き要素の格納だけができる。断面全体の格納はできない。
選択要素	.allのallキーワードの使用も含めて，完全にサポートしている。
リテラル	nullキーワードも含めて，完全にサポートしている。
ブール・シンボル	完全にサポートしている (TRUE， FALSE)。
集合体	EXAMINE コマンドでレコード全体および配列のオブジェクトを確認することができる。配列またはレコードの構成要素に値を格納することができる。文字列の値を格納する場合を除き，DEPOSIT コマンドで集合体を使用することはできない。

C.3.1.2 定義済みの属性

サポートされている Ada の定義済みの属性を次に示します。デバッガの SHOW SYMBOL/TYPE コマンドで得られる情報は，属性 P′FIRST， P′LAST，P′LENGTH， P′SIZE，および P′CONSTRAINED によって得られる情報と同じですので注意してください。

属性	デバッガのサポート
P′CONSTRAINED	接頭辞 P が，判別子の付いたレコード・オブジェクトを表す場合。P′CONSTRAINED の値は， P の現在の状態 (制約付き，または制約なし) を表す。
P′FIRST	接頭辞 P が列挙型または列挙型の部分型を表す場合。 P の下限を示す。
P′FIRST	接頭辞 P が配列型に対応しているか，制約付き配列の部分型を表す場合。最初の添字範囲の下限を示す。
P′FIRST(N)	接頭辞 P が配列型に対応しているか，制約付き配列の部分型を表す場合。 N 番目の添字範囲の下限を示す。
P′LAST	接頭辞 P が列挙型または列挙型の部分型を表す場合。 P の上限を示す。
P′LAST	接頭辞 P が配列型に対応しているか，制約付き配列の部分型を表す場合。最初の添字範囲の上限を示す。
P′LAST(N)	接頭辞 P が配列型に対応しているか，制約付き配列の部分型を表す場合。 N 番目の添字範囲の上限を示す。

属性	デバッガのサポート
P′LENGTH	接頭辞 P が配列型に対応しているか，制約付き配列の部分型を表す場合。最初の添字範囲の値の個数を示す (範囲が空の場合はゼロ)。
P′LENGTH(N)	接頭辞 P が配列型に対応しているか，制約付き配列の部分型を表す場合。 N 番目の添字範囲の値の個数を示す (範囲が空の場合はゼロ)。
P′POS(X)	接頭辞 P が列挙型または列挙型の部分型を表す場合。値 X の位置番号を示す。最初の位置番号は 0。
P′PRED(X)	接頭辞 P が列挙型または列挙型の部分型を表す場合。位置番号が X より 1 小さい P 型の値を示す。
P′SIZE	接頭辞 P がオブジェクトを示す場合。オブジェクトを保持するために割り当てられたビット数を示す。
P′SUCC(X)	接頭辞 P が列挙型または列挙型の部分型を表す場合。位置番号が X より 1 大きい P 型の値を示す。
P′VAL(N)	接頭辞 P が列挙型または列挙型の部分型を表す場合。位置番号が N の P 型の値を示す。最初の位置番号は 0。

C.3.1.2.1 列挙型の属性の指定

次の宣言について考えます。

```
type DAY is
   (MONDAY,TUESDAY,WEDNESDAY,THURSDAY,FRIDAY,SATURDAY,SUNDAY);
MY_DAY : DAY;
```

列挙型の属性の使用例を次に示します。属性の値を確認するのに EXAMINE コマンドを使用することはできませんので注意してください。これは属性が変数名ではないためです。属性の値を確認するときは，代わりに EVALUATE コマンドを使用してください。これと同じ理由から， DEPOSIT コマンドでは属性を:=演算子の右辺にしか置けません。

```
DBG> EVALUATE DAY'FIRST
MON
DBG> EVALUATE DAY'POS(WEDNESDAY)
2
DBG> EVALUATE DAY'VAL(4)
FRI
DBG> DEPOSIT MY_DAY := TUESDAY
DBG> EVALUATE DAY'SUCC(MY_DAY)
WED
DBG> DEPOSIT . := DAY'PRED(MY_DAY)
DBG> EXAMINE .
EXAMPLE.MY_DAY: MONDAY
DBG> EVALUATE DAY'PRED(MY_DAY)
%DEBUG-W-ILLENUMVAL, enumeration value out of legal range
```

C.3.1.2.2 オーバーロードされた列挙リテラルの解決

次の宣言について考えます。

```
type MASK is (DEC,FIX,EXP);
type CODE is (FIX,CLA,DEC);
MY_MASK : MASK;
MY_CODE : CODE;
```

次の例では，オーバーロードされた列挙リテラル FIX を型明示式 CODE'(FIX) で解決します。この列挙リテラル FIX は CODE 型と MASK 型の両方に属しています。

```
DBG> DEPOSIT MY_CODE := FIX
%DEBUG-W-NOUNIQUE, symbol 'FIX' is not unique
DBG> SHOW SYMBOL/TYPE FIX
data EXAMPLE.FIX
    enumeration type (CODE, 3 elements), size: 1 byte
data EXAMPLE.FIX
    enumeration type (MASK, 3 elements), size: 1 byte
DBG> DEPOSIT MY_CODE := CODE'(FIX)
DBG> EXAMINE MY_CODE
EXAMPLE.MY_CODE:        FIX
```

C.3.2 演算子と式

次の節では， Ada の演算子と言語式に関するデバッガのサポートについて説明します。

C.3.2.1 言語式の演算子

サポートされている Ada の言語式の演算子を次に示します。

種類	シンボル	機能
接頭辞	+	単項正符号 (一致)
接頭辞	–	単項負符号 (否定)
挿入辞	+	加算
挿入辞	–	減算
挿入辞	*	乗算
挿入辞	/	除算
挿入辞	MOD	モジュロ
挿入辞	REM	剰余
接頭辞	ABS	絶対値
挿入辞	&	連結 (文字列型のみ)
挿入辞	=	等値 (スカラ型および文字列型のみ)
挿入辞	/=	不等 (スカラ型および文字列型のみ)
挿入辞	>	大なり (スカラ型および文字列型のみ)
挿入辞	>=	以上 (スカラ型および文字列型のみ)
挿入辞	<	小なり (スカラ型および文字列型のみ)
挿入辞	<=	以下 (スカラ型および文字列型のみ)
接頭辞	NOT	論理否定
挿入辞	AND	論理積 (ビット配列以外の場合)
挿入辞	OR	論理和 (ビット配列以外の場合)
挿入辞	XOR	排他的論理和 (ビット配列以外の場合)

次に示す項目はサポートされていません。

- 配列全体の演算，レコード全体の演算
- and thenおよびor elseの短絡制御形式
- inおよびnot inのメンバシップ・テスト
- ユーザ定義の演算子

C.3.2.2 言語式

サポートされている Ada の式を次に示します。

式の種類	デバッガのサポート
型変換	Ada で指定された**明示的な**型変換はどれもサポートしていない。しかし，デバッガが式を評価するときは数値型の間で特定の型変換が**暗黙に**行われる。 精度の異なる型が式に含まれている場合，デバッガは式を評価する前に，精度の低い型を精度の高い型に変換する。 • 整数型と浮動小数点数型が混在しているときは，整数型が浮動小数点数型に変換される。 • 整数型と固定小数点数型が混在しているときは，整数型が固定小数点数型に変換される。 • サイズの異なる整数型 (たとえば，バイト整数とワード整数) が混在しているときは，サイズの小さい整数型が大きい整数型に変換される。
部分型	完全にサポートしている。ただし，デバッガは，部分型および範囲の制約がある型を“部分範囲”型として示す。
型明示式	オーバーロードされた列挙リテラル (識別子は同じだが，異なる列挙型に含まれるリテラル) を解決しなければならないので，サポートしている。それ以外の目的では型明示式をサポートしていない。
アロケータ	アロケータを使用する演算は何もサポートしていない。
ユニバーサル式	サポートしていない。

C.3.3 データ型

サポートされている Ada のデータ型を次に示します。

Ada **のデータ型**	VMS **のデータ型名**
INTEGER	ロングワード整数(L)
SHORT_INTEGER	ワード整数(W)
SHORT_SHORT_INTEGER	バイト整数(B)
SYSTEM.UNSIGNED_QUADWORD	クォドワード符号なし (QU)
SYSTEM.UNSIGNED_LONGWORD	ロングワード符号なし (LU)
SYSTEM.UNSIGNED_WORD	ワード符号なし (WU)
SYSTEM.UNSIGNED_BYTE	バイト符号なし (BU)

Ada のデータ型	VMS のデータ型名
FLOAT	F 浮動小数点数(F)
SYSTEM.F_FLOAT	F 浮動小数点数(F)
SYSTEM.D_FLOAT	D 浮動小数点数(D)
LONG_FLOAT	LONG_FLOAT (D_FLOAT) プラグマが有効のときは， D 浮動小数点数(D)。 LONG_FLOAT (G_FLOAT) プラグマが有効のときは， G 浮動小数点数(G)。
SYSTEM.G_FLOAT	G 浮動小数点数(G)
IEEE_SINGLE_FLOAT (Alpha 固有)	S 浮動小数点数(FS)
IEEE_DOUBLE_FLOAT (Alpha 固有)	T 浮動小数点数(FT)
固定	(なし)
STRING	ASCII テキスト(T)
BOOLEAN	境界合わせされたビット列(V)
BOOLEAN	境界合わせされていないビット列 (VU)
列挙	値が符号なしバイトと一致する列挙型は，バイト符号なし (BU)。値が符号なしワードと一致する列挙型は，ワード符号なし (WU)。それ以外の列挙型に対応するオペレーティング・システムのデータ型はない。
配列	(なし)
レコード	(なし)
アクセス (ポインタ)	(なし)
タスク	(なし)

C.3.4 コンパイルとリンク

ユーザのシステムのプログラム・ライブラリ ADA$PREDEFINED にある， Ada の定義済みユニットは/NODEBUG 修飾子を使用してコンパイルされています。定義済みユニットの中で宣言されている名前をデバッガで参照するにはその前に，定義済みユニットのソース・ファイルを ACS EXTRACT SOURCE コマンドでコピーしなければなりません。そして，コピーしたファイルを/DEBUG 修飾子を使用してコンパイルし，適切なライブラリにしてから，/DEBUG 修飾子を付けてプログラムを再リンクする必要があります。

Ada の各コンパイル・コマンドで/NODEBUG 修飾子を使用した場合，デバッグのためにモジュールに含まれるのはグローバル・シンボル・レコードだけです。このときのグローバル・シンボルは，プログラムが他言語のモジュールへエクスポートする名前です。プログラムは次の Ada のエクスポート・プラグマを使用して名前をエクスポートします。

```
EXPORT_PROCEDURE
EXPORT_VALUED_PROCEDURE
EXPORT_FUNCTION
EXPORT_OBJECT
```

EXPORT_EXCEPTION
PSECT_OBJECT

ACS LINK コマンドで/DEBUG 修飾子を使用すると，リンカは，実行可能なイメージ内の特定のユニットのクロージャにデバッグの情報をすべて含めます。

C.3.5 ソースの表示

Ada プログラム固有の次の理由から，ソース・コードを表示できないことがあります。

- Ada の初期化コードまたは確立コードで実行が一時停止される。この初期化コードや確立コードのために，ソース・コードを使用することができない。
- コピーされたソース・ファイルが，元のコンパイル単位をコンパイルするのに使用したプログラム・ライブラリの中に含まれていない。
- 外部ソース・ファイルが，そのコンパイル単位が当初コンパイルされたときにあった場所にない。
- 実行可能なイメージを生成してからソース・ファイルの修正が行われており，コピーされたソース・ファイルや外部ソース・ファイルはすでに存在しない。

Ada プログラムでソース・コードの表示を制御する方法を次に説明します。

コンパイラ・コマンドの/COPY_SOURCE 修飾子 (省略時設定) が有効な状態でプログラムをコンパイルした場合，表示される Ada のソース・コードは，プログラムを新しくコンパイルしたときのプログラム・ライブラリにあるソース・ファイルをコピーして取得されます。/NOCOPY_SOURCE 修飾子を使用してプログラムをコンパイルした場合，表示されるソース・コードは，プログラムのコンパイル単位に対応する外部ソース・ファイルから取得されます。

コピーされるソース・ファイルのファイル指定，または外部ソース・ファイルのファイル指定は，対応するオブジェクト・ファイルに埋め込まれています。たとえば，コンパイル単位をコピーするのに ACS の COPY UNIT コマンドを使用しても，あるいは，ライブラリ全体をコピーするのに DCL の COPY コマンドか BACKUP コマンドを使用しても，デバッガはコピーされたソース・ファイルから元のプログラム・ライブラリを検索します。コピー後に元のコンパイル単位を修正した場合，またはコピー後に元のライブラリを削除した場合，デバッガはコピーされた元のソース・ファイルを見つけることができません。同様に，外部ソース・ファイルを別のディスクやディレクトリに移動したときも，デバッガは元のソース・ファイルを見つけられません。

このような場合は，ソースの表示のための正しいファイルを検索するために， SET SOURCE コマンドを使用します。プログラム・ライブラリやソース・コードの複数のディレクトリを示す検索リストを指定することができます。次に例を示します。ADA$LIB は，プログラム・ライブラリ・マネージャが現在のプログラム・ライブラリと等しく定義する論理名です。

```
DBG> SET SOURCE ADA$LIB,DISK:[SMITH.SHARE.ADALIB]
```

ユーザがデバッガの EDIT コマンドを使用するときにデバッガが取り出す外部ソース・ファイルを検索するための検索リストは，SET SOURCE コマンドでは変更されません。ソース・ファイルを検索する場所を EDIT コマンドで指定するときは，SET SOURCE/EDIT コマンドを使用します。

C.3.6 EDIT コマンド

Ada プログラムで省略時にデバッガの EDIT コマンドが取り出す外部ソース・ファイルは，実行を現在一時停止しているコンパイル単位を作成するためにコンパイルされた外部ソース・ファイルです。プログラム・ライブラリ内のコピーされたソース・ファイルは編集しないでください。デバッガはこのソース・ファイルをソースの表示のために使用します。

編集するソース・ファイルのファイル指定は，コンパイル中に，対応するオブジェクト・ファイルに埋め込まれます (/NODEBUG を指定しなかった場合)。コンパイル後にソース・ファイルの配置場所を変更すると，デバッガがそれらのソース・ファイルを見つけられなくなります。

配置場所を変更したときは，デバッガの SET SOURCE/EDIT コマンドを使用すると，デバッガがソース・ファイルを検索する，1つまたは複数のディレクトリからなる検索リストを指定することができます。次に例を示します。

```
DBG> SET SOURCE/EDIT [],USER:[JONES.PROJ.SOURCES]
```

SET SOURCE/EDIT コマンドを使用しても，デバッガがソースの表示に使用するコピー元のソース・ファイルを検索するための検索リストは変更されません。

SHOW SOURCE/EDIT コマンドを使用すると，現在 EDIT コマンドで使用中のソース・ファイル検索リストが表示されます。CANCEL SOURCE/EDIT コマンドを使用すると，現在 EDIT コマンドで使用中のソース・ファイル検索リストが取り消され，省略時の検索モードに戻ります。

C.3.7 GO コマンドと STEP コマンド

Ada プログラムで GO コマンドや STEP コマンドを使用する場合は，次の点に注意してください。

- コンパイラ生成初期化コード内の命令をステップ実行しないように するには，デバッグ・セッションを開始するときに，STEP コマンド ではなく GO コマンドを使用してください。
  - メイン・プログラムの開始時に初期化コードとパッケージ 確立コードをとばして，定義済みブレークポイントに直行 するには，GO コマンドを使用します。

– メイン・プログラムが開始される前の，ライブラリ・パッケージ の確立開始時点で実行を一時停止するには， GO コマンドと ブレークポイントを使用します。

パッケージ確立フェーズを監視する方法については， オンライン・ヘルプの Debugging_Ada_Library_Packages を参照 してください。

- 1 行に複数の文がある場合， STEP コマンドは，その行の文すべてを 1 つのステップの一部分として実行します。
- タスク・エントリ呼び出しとサブプログラム呼び出しは異なります。 タスク・エントリ呼び出しはキューに登録されるのであって， ただちに実行されるとは限らないからです。 タスク・エントリ呼び出しに実行を移すのに STEP コマンドを使用 すると，期待どおりの結果が得られないことがあります。

C.3.8 Ada のライブラリ・パッケージのデバッグ

Ada のメイン・プログラム，または Ada のコードを呼び出す Ada 以外のメイン・プログラムが実行されるときに， Ada の実行時ライブラリに対して初期化コードが実行され，プログラムが依存するすべてのライブラリ単位に対して確立コードが実行されます。確立コードは，メイン・プログラムの実行前にライブラリ単位を適切な順序で確立します。ライブラリの仕様と本体，およびそれらのサブユニットもこのプロセスによって確立されます。

ライブラリ・パッケージを確立すると，次の処理が行われます。

- パッケージの宣言を有効にする。
- 宣言に初期化コードを含む変数を初期化する。
- パッケージ本体のbegin文とend文の間に ある一連の文をすべて実行する。

Ada プログラムをデバッガの制御下に置くと，初めに，初期化コードの実行前およびライブラリ単位の確立前に実行が一時停止されます。次に例を示します。

```
DBG> RUN FORMS
Language: ADA, Module: FORMS
Type GO to reach main program
DBG>
```

この時点で， GO と入力してメイン・プログラムを開始する前に，注目したいパッケージの仕様や本体にブレークポイントを設定することによって，ライブラリ・パッケージ内の命令を何箇所かステップ実行してチェックすることができます。その後，各パッケージを開始するために GO コマンドを使用します。パッケージの本体にブレークポイントを設定するには， SET BREAK コマンドでパッケージ単位名を指定します。パッケージの仕様にブレークポイントを設定するときは，パッケージ単位名をその後ろにアンダスコア(_)を付けて指定します。

パッケージの本体にブレークポイントを設定しても，その本体に対するデバッガのモジュールが設定されていないときはブレークされません。モジュールが設定されていない場合は，パッケージの仕様のところでブレークされます。このようになるのは，with 句の中で名前を指定されているパッケージの仕様に対するモジュールを，デバッガが自動的に設定するためです。デバッガは，対応するパッケージ本体に対するモジュールを自動的には設定しません (第 C.3.14 項を参照)。

また，パッケージの仕様の中で宣言されているサブプログラムにブレークポイントを設定するには，パッケージの本体に対するモジュールをユーザが設定する必要があります。

コンパイラは，ライブラリ・パッケージの中で宣言されているサブプログラムに固有の名前を作成しますので注意してください。これらの名前は，オーバーロードされた名前です。または，オーバーロードされた名前になることもあります。デバッガの出力には，これらの固有の名前が使用されます。また，使用しないと名前があいまいになる場合は，この固有の名前を各コマンドで使用する必要があります。オーバーロードされた名前とシンボルの解決についての詳しい説明は，第 C.3.15 項を参照してください。

C.3.9 定義済みブレークポイント

Ada プログラムまたは Ada のコードを呼び出す Ada 以外のプログラムでデバッガを起動すると， Ada のタスキング例外イベントに対応する 2 つのブレークポイントが自動的に設定されます。これらのブレークポイントは， Ada の実行時ライブラリが存在しているときに，デバッガの初期化中に自動的に設定されます。

この状況で SHOW BREAK コマンドを入力すると，次のようにブレークポイントが表示されます。

```
DBG> SHOW BREAK
Predefined breakpoint on ADA event "EXCEPTION_TERMINATED"
    for any value
Predefined breakpoint on ADA event "DEPENDENTS_EXCEPTION"
    for any value
DBG>
```

C.3.10 例外の監視

Ada プログラムの場合，デバッガは次の 3 種類の例外を認識します。

- ユーザ定義の例外 — Ada のコンパイル単位内で， Ada の予約語 exceptionで宣言された例外
- Ada の定義済みの例外。 PROGRAM_ERROR または CONSTRAINT_ERROR など。
- その他の (Ada 以外の) すべての例外または VMS 条件

次の各サブトピックでは，これらの例外を監視する方法について説明します。

C.3.10.1 すべての例外の監視

SET BREAK/EXCEPTION コマンドを使用すると，どの例外または VMS 条件にもブレークポイントを設定することができます。これには， Ada の実行時ライブラリの内部でシグナル通知される特定の VMS 条件も含まれます。これらの条件は言語処理系の機構によるものであり，プログラムの障害を意味するものではありません。また，これらの条件を Ada の例外ハンドラで処理することはできません。プログラムをデバッグ中にこれらの条件が現れる場合は，ブレークポイントを設定するときに，例外の種類を指定することもできます。

SET TRACE/EXCEPTION コマンドの結果， Ada の CONSTRAINT_ERROR 例外のためにトレースポイントが現れるときの例を，次に示します。

```
DBG> SET TRACE/EXCEPTION
DBG> GO
      . . .
%ADA-F-CONSTRAINT_ERRO, CONSTRAINT_ERROR
-ADA-I-EXCRAIPRI, Exception raised prior to PC = 00000A7C
trace on exception preceding
    ADA$RAISE\ADA$RAISE_CONDITION.%LINE 333+12
      . . .
```

内部で例外が発生したサブプログラムを呼び出したとき，または例外発生の対象となったサブプログラムを呼び出したときのトレースバックを SHOW CALLS コマンドで表示する例を次に示します。

```
DBG> SET BREAK/EXCEPTION DO (SHOW CALLS)
DBG> GO
       . . .
%SYSTEM-F-INTDIV, arithmetic trap, integer divide
     by zero at PC=000008AF,
PSL=03C000A2 break on exception preceding
     SYSTEM_OPS.DIVIDE.%LINE 17+6
     17:      return X/Y;
 module name     routine name       line    rel PC    abs PC
*SYSTEM_OPS      DIVIDE               17   00000015  000008AF
*PROCESSOR       PROCESSOR            19   000000AE  00000BAD
*ADA$ELAB_PROCESSOR
                 ADA$ELAB_PROCESSOR        00000009  00000809
                 LIB$INITIALIZE            00000054  00000C36
 SHARE$ADARTL                              00000000  000398BE
*ADA$ELAB_PROCESSOR
                 ADA$ELAB_PROCESSOR        0000001B  0000081B
                 LIB$INITIALIZE            0000002F  00000C21
```

この例では， SYSTEM_OPS パッケージの DIVIDE サブプログラムの 17 行目で SS$_INTDIV 条件が発生します。 SS$_INTDIV などのいくつかの条件は Ada のある種の定義済みの例外と等価なものとして処理されるという重要な働きを，この例は示しています。

Ada の定義済みの例外と条件を照合するのは，例外部分を持つすべてのフレーム用に Ada に備えられている条件ハンドラです。したがって， Ada の例外名と等価な名前を持った条件によって例外のブレークポイントまたはトレースポイントが検出された場合，メッセージには，システムの条件コード名**だけ**が表示され，対応する Ada の例外名は表示されません。

C.3.10.2 特定の例外の監視

例外が発生すると，デバッガは次の組み込みシンボルを設定します。この組み込みシンボルを使用すると，特定の例外が発生したときだけ検出されるように，例外のブレークポイントやトレースポイントを指定することができます。

%EXC_FACILITY	例外を生じたファシリティの名前を示す文字列。 Ada の定義済みの例外およびユーザ定義の例外のファシリティ名は， ADA。
%EXC_NAME	例外名を示す大文字の文字列。発生した例外が Ada の定義済みの例外の場合， 15 文字を超える分の名前は切り捨てられる。たとえば，CONSTRAINT_ERROR は切り捨てられ， CONSTRAINT_ERRO になる。発生した例外がユーザ定義の例外の場合，%EXC_NAME には "EXCEPTION"という文字列が入り，ユーザ定義の例外名は%ADAEXC_NAME に入る。
%ADAEXC_NAME	発生した例外がユーザ定義の例外の場合，%ADAEXC_NAME にはその例外名を示す文字列が入り，%EXC_NAME には "EXCEPTION"という文字列が入る。発生した例外がユーザ定義の例外ではない場合，%ADAEXC_NAME には空文字列が入り，%EXC_NAME に例外名が入る。
%EXC_NUM	例外の個数。
%EXC_SEVERITY	例外の重大度を示す文字列 (F， E， W， I， S，または?)。

C.3.10.3 処理される例外と例外ハンドラの監視

SET BREAK/EVENT コマンドと SET TRACE/EVENT コマンドを使用すると，Ada の例外ハンドラが処理しようとしている例外にブレークポイントとトレースポイントを設定することができます。制御を渡される Ada の各例外ハンドラの実行を，これらのコマンドで監視することができます。

これらのコマンドでは，次の 2 つのイベント名を指定することができます。

HANDLED	何らかの Ada の例外ハンドラが例外を処理しようとしているときに検出される。 (HANDLED_OTHERSを含む)
HANDLED_OTHERS	Otherを選択している Ada の例外ハンドラが例外を処理しようとしているときのみ検出される。

たとえば，次のコマンドは， Ada の例外ハンドラが例外を処理しようとしているときに検出されるブレークポイントを設定します。

```
DBG> SET BREAK/EVENT=HANDLED
```

ブレークポイントが検出されると，処理されようとしている例外と実行されようとしている例外ハンドラをデバッガが示します。その後，この情報を利用して特定のハンドラにブレークポイントを設定することができます。また， GO コマンドを入力して， Ada のどのハンドラが次に例外を処理しようとするかを見ることもできます。次に例を示します。

```
DBG> GO
     . . .
break on Ada event HANDLED
  task %TASK 1 is about to handle an exception
  The Ada exception handler is at: PROCESSOR.%LINE 21
    %ADA-F-CONSTRAINT_ERRO, CONSTRAINT_ERROR
    -ADA-I-EXCRAIPRI, Exception raised prior to PC = 00000A7C
DBG> SET BREAK PROCESSOR.%LINE 21; GO
```

C.3.11 データの検査と操作

データを検査または操作するときは，次の関連事項に注意してください。

- 非静的変数 (ライブラリ・パッケージの中で宣言されていない すべての変数) の確認，または非静的変数への格納を行うには， その前に，非静的変数を定義するサブプログラムやタスクなどが 呼び出しスタック上でアクティブになっていなければなりません。
- Ada のサブプログラムの仮パラメータまたは Ada の変数を確認したり， 格納したり，または評価したりするには，その前にそのパラメータや 変数が確立されていなければなりません。つまり，パラメータや変数 の宣言の後ろへステップ実行するか，または実行を移さなければなり ません。宣言が確立されていない変数や仮パラメータに格納されて いる値は，違う場合があります。

型明示式の使用も含め，デバッガではほとんどの場合，プログラムのソース・コードに指定するのと全く同じように，変数や式をデバッガ・コマンドの中で指定することができます。次の各サブトピックでは，レコードとアクセス型についてのデバッガのサポートに関する補足事項について説明します。

C.3.11.1 レコード

デバッガによるレコードのサポートについては次の点に注意してください。

- レコード (シンボル) 宣言と型宣言の有効範囲が異なっている 場合， Ada の特定のレコード変数では，デバッガがレコードの 構成要素を正確に表示できず，エラー・メッセージ NOACCESSR が表示されることがあります。
- 可変レコードについては，アクティブでない可変部の構成要素に値を 代入したり，その構成要素の値を確認することができます。 しかしこれは Ada では不正な動作になるので，情報メッセージも あわせて発行されます。たとえば， REC1 レコードに STATUS という 名前の可変フィールドがあり， STATUS の値が REC1.COMP3 が非アクティブであることを示しているとすると， 次のようになります。

```
DBG> EXAMINE REC1.COMP3
%DEBUG-I-BADDISCVAL, incorrect value of 1 in discriminant
    field STATUS
MAIN.REC1.COMP3:    438
```

C.3.11.2 アクセス型

デバッガによるアクセス型のサポートについては，次の点に注意してください。

- デバッガはアロケータをサポートしていないので，デバッガで新しい アクセス・オブジェクトを作成することはできない。
- EXAMINE コマンドでアクセス・オブジェクトの名前を指定すると， デバッガはその名前が指すオブジェクトのメモリ記憶位置を表示する。
- 指定オブジェクトの値を確認するには，選択要素の表記法を使用して .ALL を指定しなければならない。 たとえば， A が指すレコード・アクセス・オブジェクトの値を確認する には，次のようにする。

```
DBG> EXAMINE A.ALL
EXAMPLE.A.ALL
    NAME(1..10):        "John Doe  "
    AGE :       6
    NEXT:       1462808
```

- 指定オブジェクトの構成要素の 1 つを確認する場合は，選択要素の 構文から.ALL を省略することができる。次に例を示す。

```
DBG> EXAMINE A.NAME
EXAMPLE.A.ALL.NAME(1..10):     "John Doe  "
```

不完全型についてのデバッガのサポートの例を示します。次の宣言を見てください。

```
package P is
   type T is private;
private
   type T_TYPE;
   type T is access T_TYPE;
end P;

package body P is
   type T_TYPE is
      record
        A: NATURAL := 5;
        B: NATURAL := 4;
      end record;

   T_REC: T_TYPE;
   T_PTR: T := new T_TYPE'(T_REC);
end P;

with P; use P;
procedure INCOMPLETE is
   VAR: T;
begin
    . . .
end INCOMPLETE;
```

T 型についての完全な情報がデバッガに与えられていないので， VAR 変数を操作することはできません。ただし，パッケージ本体 P の中で宣言されているオブジェクトについては情報が与えられているので， T_PTR 変数と T_REC 変数を操作することはできます。

C.3.12 モジュール名とパス名

Ada のデバッガ・モジュール名は，対応するコンパイル単位の名前と同じです。これには次の条件があります。あいまいさをなくすために，仕様の名前にアンダスコア文字(_)を追加して仕様と本体の名前とを区別してください。たとえば， TEST (本体)， TEST_ (仕様) とします。プログラム内のモジュールの正確な名前を確認するには， SHOW MODULE コマンドを使用します。

ほとんどの場合，パス名を指定する際に，仕様と本体とを区別するアンダスコアを後部に入力する必要はありません。通常，デバッガは状況に応じてこの 2 つを区別します。したがって，あいまいさを解消しなければならないときだけ，この命名規則を使用してください。

デバッガの言語が Ada に設定されている場合，デバッガは，パス名の要素を区切るのに選択要素の表記法を一般的に使用し， Ada の規則に従ってパス名を構成します。他の言語では要素を区切るのにバックスラッシュを使用します。次に例を示します。

```
TEST_.A1        ! A1 はコンパイル単位 TEST のパッケージ
                ! 仕様の中で宣言されている
TEST.B1         ! B1 はコンパイル単位 TEST のパッケージ
                ! 本体の中で宣言されている
```

サブユニットのパス名 (展開される名前) は，最大 247 文字の長さまで指定することができます。

シンボルがパッケージの外部で直接に可視になるよう，パッケージ内で use 句によって宣言されている場合は，プログラム自体またはデバッガ・コマンドのいずれにおいても，そのシンボルを参照するために，展開された名前 (*package-name.symbol*) を指定する必要はありません。

指定されたブロック，サブプログラム，またはパッケージが use 句の中で参照しているパッケージを示すには， SHOW SYMBOL/USE_CLAUSE コマンドを使用します (指定されているパッケージがライブラリであるかどうかは関係ありません)。指定されている要素が，ライブラリであるかどうかにかかわらずパッケージである場合は，use 句内の特定のモジュール名を示すブロック，サブプログラム，パッケージなどもこのコマンドで示すことができます。次に例を示します。

```
DBG> SHOW SYMBOL/USE_CLAUSE B_
package spec B_
    used by:  F
    uses:     A_
```

ソース・コードのループ文または宣言ブロックにラベルが付けられている場合，デバッガはそのラベルを表示します。それ以外の場合，**ループ**文についてはLOOP$*n*を表示し，**宣言**ブロックについてはBLOCK$*n*を表示します。それぞれの*n*は，その文またはブロックが始まる行番号を示します。

C.3.13 シンボル検索規則

Adaプログラムでパス名(Adaの展開された名前を含む)を指定しなかった場合，デバッガは次のように実行時シンボル・テーブルを検索します。

1. デバッガは，現在そこで実行を一時停止している現在のPC値に近い ブロックまたはルーチンの中でシンボルを検索します。
2. シンボルが見つからない場合は， use句に指定されているパッケージ を次に検索します。デバッガは，現在の有効範囲の領域と同じ モジュールに宣言があるパッケージと，ライブラリ・パッケージとを 区別しません。可視である複数のパッケージの中で同じシンボルが 宣言されている場合， Adaの規則によりシンボルは固有のものでは なくなり，次のようなメッセージが発行されます。

   ```
   %DEBUG-E-NOUNIQUE, symbol 'X' is not unique
   ```

3. それでもまだシンボルが見つからない場合，他の言語については， 呼び出しスタックと他の有効範囲を検索します。

C.3.14 モジュールの設定

ユーザまたはデバッガがAdaのモジュールを設定するときに，省略時設定では，デバッガが“関連”モジュールもすべて設定します。この関連モジュールとは，設定されるモジュールの中でシンボルが可視でなければならないモジュールを指します。これらの関連モジュールは， with句またはサブユニット関係のどちらかで設定されるモジュールに関連づけられます。

関連モジュールの設定は次のように行われます。設定されるモジュールがM1である場合，次の各モジュールが関連しているとみなされ，設定されます。

- M1が**ライブラリ本体**である場合に，対応する ライブラリ仕様があれば，それも設定されます。
- M1が**サブユニット**である場合， その親ユニットと，親ユニットのすべての親も設定されます。
- M1がwith句で**ライブラリ・パッケージ** P1を指定している場合， P1の仕様も設定されます。 P1の本体とP1のサブユニットの中で宣言されているシンボルは 外部では可視でないので， P1の本体とP1のサブユニットは何も 設定されません。

P1 の仕様が with 句で P2 パッケージを指定している場合は， P2 の仕様も設定されます。同様に， P2 の仕様が with 句で P3 パッケージを指定している場合は P3 の仕様も設定され， 同じように設定されていきます。 このようなライブラリ・パッケージはすべて，それらの仕様が設定 されるので，他のパッケージで宣言されているデータ構成要素 (たとえばレコードの構成要素) をアクセスすることができます。

- M1 が with 句で**ライブラリのサブプログラム** を指定している場合，そのサブプログラムは **設定されません**。 M1 の中で可視でなければなら ないのはサブプログラム名だけです。ライブラリのサブプログラム内 の宣言は，そのサブプログラムの外部で可視である必要はありません。 したがって，デバッガがライブラリのサブプログラム名を RST に挿入 するのは， M1 が設定されるときになります。

多くのモジュールが設定されることでデバッガの性能に問題が生じるときは， SET MODE NODYNAMIC コマンドを使用してください。このコマンドで，動的モジュールの設定が禁止されるとともに，関連モジュールの設定が禁止されます。そのあとで SET MODULE コマンドを使用して，個々のモジュールを明示的に設定する必要があります。

省略時の SET MODULE コマンドでは，コマンドで指定したモジュールと同時に関連モジュールが設定されます。

SET MODULE/NORELATED コマンドは，明示的に指定されたモジュールのみを設定します。ただし， SET MODULE/NORELATED を使用すると，別のユニットで宣言されており，実行の時点で可視状態になっているべきシンボルが可視状態でなくなっていたり，同じシンボルの再宣言によって隠蔽されているべきシンボルが可視状態になっていたりすることがあります。

CANCEL MODULE/NORELATED コマンドで RST から削除されるのは，明示的に指定したモジュールだけです。省略時設定の CANCEL MODULE/NORELATED コマンドは， Ada の有効範囲規則と可視性の規則の目的に合致した方法で関連モジュールを削除します。正確な効果はモジュールの関連性に依存します。

サブユニットの関連と直接関連の違いは，ライブラリ・パッケージの関連の違いと同様です。

C.3.14.1 パッケージ本体のためのモジュールの設定

デバッガは，パッケージ本体のためのモジュールを自動的には設定しません。

パッケージ本体をデバッグするには，または対応するパッケージ仕様の中で宣言されているサブプログラムをデバッグするには，ユーザが自分でライブラリ・パッケージ本体のためのモジュールを設定しなければならない場合があります。

C.3.15 オーバーロードされた名前とシンボルの解決

オーバーロードされた名前やシンボルが現れると，次のようなメッセージが表示されます。

```
%DEBUG-E-NOTUNQOVR, symbol 'ADD' is overloaded
     use SHOW SYMBOL to find the unique symbol names
```

オーバーロードされたシンボルが列挙リテラルである場合は，オーバーロードを解決するために，型明示式を使用することができます。

オーバーロードされたシンボルがサブプログラムまたはタスクの accept 文を表す場合は，コンパイラがデバッガ用に作成する固有の名前を使用することができます。名前は，あとでパッケージ本体でオーバーロードされることがあるので，コンパイラは必ず各サブプログラム固有の名前をライブラリ・パッケージの仕様に作成します。タスクの accept 文やサブプログラムが他の場所で宣言されているときは，タスクの accept 文やサブプログラムが実際にオーバーロードされる場合にだけ，固有の名前が作成されます。

オーバーロードされたタスクの accept 文とサブプログラムの名前は， 2 つのアンダスコアとそれに続く整数からなる接尾辞によって区別されます。それぞれのシンボルは，この整数で別々のものとして示されます。名前がオーバーロードされているサブプログラムを別々のものとして指定するには，固有の名前を表記してデバッガ・コマンドを使用する必要があります。しかし，あいまいさがない場合は，固有の名前が作成されてもその名前を使用する必要はありません。

C.3.16 CALL コマンド

Ada プログラムで確実に CALL コマンドを使用できるのは，エクスポートされているサブプログラムでこのコマンドを使用する場合だけです。エクスポートされるサブプログラムは，ライブラリのサブプログラムであるか，またはライブラリ・パッケージの最も外側にある宣言部で宣言されている必要があります。

CALL コマンドは，サブプログラムをエクスポートできるかどうかはチェックしません。また，ユーザが指定するパラメータの受け渡し方式もチェックしません。パラメータの値を変更するために CALL コマンドを使用することはできませんので注意してください。

Ada の実行時ライブラリを実行中に CALL コマンドを入力するとデッドロックを起こすことがあります。実行時ライブラリ・ルーチンは，タスキング環境で実行時ライブラリ・ルーチンを作動可能にする内部ロックを取得および解放します。デッドロックを生じる可能性があるのは， CALL コマンドから呼び出されたサブプログラムが，実行中の実行時ライブラリ・ルーチンによってロックされている資源を要求するときです。非タスキング・プログラムにおいてこういった状況が生じないようにするには， Ada の文を実行する直前，または実行した直後に CALL コマンドを入力して

ください。しかし，この方法では，タスキング・プログラムでのデッドロックの発生を完全に防ぐことはできません。呼び出しの時点で他の何らかのタスクが実行時ライブラリ・ルーチンを実行している可能性があるからです。タスキング・プログラムでCALL コマンドを使用しなければならない場合は，呼び出されるサブプログラムがタスキング操作や入出力操作を何も実行しないようにすると，デッドロックを避けることができます。

C.4 BASIC

次の各サブトピックでは，デバッガによる BASIC のサポートについて説明します。

C.4.1 言語式の演算子

言語式でサポートされている BASIC の演算子を次に示します。

種類	シンボル	機能
接頭辞	+	単項正符号
接頭辞	–	単項負符号 (否定)
挿入辞	+	加算，文字列の連結
挿入辞	–	減算
挿入辞	*	乗算
挿入辞	/	除算
挿入辞	**	べき乗
挿入辞	^	べき乗
挿入辞	=	等値
挿入辞	<>	不等
挿入辞	><	不等
挿入辞	>	大なり
挿入辞	>=	以上
挿入辞	=>	以上
挿入辞	<	小なり
挿入辞	<=	以下
挿入辞	=<	以下
接頭辞	NOT	ビット単位の NOT
挿入辞	AND	ビット単位の AND
挿入辞	OR	ビット単位の OR
挿入辞	XOR	ビット単位の排他的論理和
挿入辞	IMP	ビット単位の含意
挿入辞	EQV	ビット単位の同値

C.4.2 言語式とアドレス式の構造

サポートされている， BASIC の言語式とアドレス式の構造を次に示します。

シンボル	構造
( )	添字指定
::	レコードの構成要素の選択

C.4.3 データ型

サポートされている BASIC のデータ型を次に示します。

BASIC のデータ型	VMS のデータ型名
BYTE	バイト整数(B)
WORD	ワード整数(W)
LONG	ロングワード整数(L)
SINGLE	F 浮動小数点数(F)
DOUBLE	D 浮動小数点数(D)
GFLOAT	G 浮動小数点数(G)
DECIMAL	パック 10 進数(P)
STRING	ASCII テキスト(T)
RFA	(なし)
RECORD	(なし)
配列	(なし)

C.4.4 デバッグのためのコンパイル

BASIC 環境でプログラムに変更を加えてから，そのプログラムを保存したり置換したりしないまま，/DEBUG 修飾子を使用してコンパイルしようとすると，“Unsaved changes,no source line debugging available” (変更を保存していないのでデバッグにソース行を使用できません) という BASIC のエラーが通知されます。この問題を防ぐには，プログラムを保存または置換してから，/DEBUG 修飾子を使用してプログラムを再コンパイルしてください。

C.4.5 定数

[*radix*]“*numeric-string*”[*type*]という形式 (“12.34”GFLOAT など) や，*n*%という形式 (整数 25 の 25%など) の BASIC の定数は，デバッガの式ではサポートされていません。

C.4.6 式の評価

BASIC 言語でオーバフローする式であっても，デバッガが評価する場合に必ずオーバフローするとはかぎりません。 BASIC の規則でオーバフローになる場合でも，デバッガは数値的に正しい結果を計算しようとします。この違いは， 10 進数を計算するときに特に影響があります。

C.4.7 行番号

デバッグ・セッションではソース・コード・ディスプレイに表示される連続した行番号を参照しますが，この行番号を生成するのはコンパイラです。 BASIC プログラムに他のファイルからコードを取り込んだり，付加したりする場合，取り込まれたコードの行にもコンパイラによって順番に行番号が付けられます。

C.4.8 ルーチンへのステップ

STEP/INTO コマンドは，外部関数をチェックするのに便利です。しかし，内部サブルーチンや DEF で実行を停止するのにこのコマンドを使用すると，デバッガは最初に実行時ライブラリ (RTL) のルーチン内の命令をステップ実行するので，有用な情報は何も得られません。次の例では， Print_routine を呼び出す 8 行目で実行が一時停止されます。

```
     . . .
->   8  GOSUB Print_routine
     9  STOP
     . . .
    20  Print_routine:
    21    IF Competition = Done
    22      THEN PRINT "The winning ticket is #";Winning_ticket
    23      ELSE PRINT "The game goes on."
    24    END IF
    25  RETURN
```

STEP/INTO コマンドを実行すると，デバッガは適切な RTL コード内の命令をステップ実行して，表示のために使用できるソース行がないことをユーザに伝えてきます。一方， STEP コマンドだけを使用すると，デバッガは Print_routine への呼び出しを通り越して，直接ソース行の 9 行目に進みます。サブルーチンまたは DEF 関数のソース・コードをチェックするには，ルーチンのラベルにブレークポイントを設定してください。たとえば， SET BREAK PRINT_ROUTINE コマンドを入力します。こうすると，正確にルーチンの開始地点この例では 20 行目で実行を一時停止してから，コード内の命令を直接ステップ実行することができます。

C.4.9 シンボル参照

単一の BASIC プログラム内の変数名とラベル名は，すべて固有のものでなければなりません。固有でない場合，デバッガはシンボルのあいまいさを解消できません。

C.5 BLISS

次の各サブトピックでは，デバッガによる BLISS のサポートについて説明します。

C.5.1 言語式の演算子

言語式でサポートされている BLISS の演算子を次に示します。

種類	シンボル	機能
接頭辞	.	間接参照
接頭辞	+	単項正符号
接頭辞	–	単項負符号 (否定)
挿入辞	+	加算
挿入辞	-	減算
挿入辞	*	乗算
挿入辞	/	除算
挿入辞	MOD	剰余
挿入辞	^	左シフト
挿入辞	EQL	等値
挿入辞	EQLU	等値
挿入辞	EQLA	等値
挿入辞	NEQ	不等
挿入辞	NEQU	不等
挿入辞	NEQA	不等
挿入辞	GTR	大なり
挿入辞	GTRU	大なり符号なし
挿入辞	GTRA	大なり符号なし
挿入辞	GEQ	以上
挿入辞	GEQU	以上符号なし
挿入辞	GEQA	以上符号なし
挿入辞	LSS	小なり
挿入辞	LSSU	小なり符号なし
挿入辞	LSSA	小なり符号なし
挿入辞	LEQ	以下
挿入辞	LEQU	以下符号なし
挿入辞	LEQA	以下符号なし

種類	シンボル	機能
接頭辞	NOT	ビット単位の NOT
挿入辞	AND	ビット単位の AND
挿入辞	OR	ビット単位の OR
挿入辞	XOR	ビット単位の排他的 OR
挿入辞	EQV	ビット単位の同値

C.5.2 言語式とアドレス式の構造

サポートされている， BLISS の言語式とアドレス式の構造を次に示します。

シンボル	構造
[ ]	添字指定
[fldname]	フィールドの選択
<p,s,e>	ビット・フィールドの選択

C.5.3 データ型

サポートされている BLISS のデータ型を次に示します。

BLISS のデータ型	VMS のデータ型名
BYTE	バイト整数(B)
WORD	ワード整数(W)
LONG	ロングワード整数(L)
QUAD (Alpha および Integrity 固有)	クォドワード(Q)
BYTE UNSIGNED	バイト符号なし(BU)
WORD UNSIGNED	ワード符号なし(WU)
LONG UNSIGNED	ロングワード符号なし(LU)
QUAD UNSIGNED (Alpha および Integrity 固有)	クォドワード符号なし(QU)
VECTOR	(なし)
BITVECTOR	(なし)
BLOCK	(なし)
BLOCKVECTOR	(なし)
REF VECTOR	(なし)
REF BITVECTOR	(なし)
REF BLOCK	(なし)
REF BLOCKVECTOR	(なし)

C.6 C

次の各サブトピックでは，デバッガによる C のサポートについて説明します。

C.6.1 言語式の演算子

言語式でサポートされている C の演算子を次に示します。

種類	シンボル	機能
接頭辞	*	間接参照
接頭辞	&	アドレス
接頭辞	sizeof	サイズ
接頭辞	–	単項負符号 (否定)
挿入辞	+	加算
挿入辞	–	減算
挿入辞	*	乗算
挿入辞	/	除算
挿入辞	%	剰余
挿入辞	<<	左シフト
挿入辞	>>	右シフト
挿入辞	= =	等値
挿入辞	!=	不等
挿入辞	>	大なり
挿入辞	>=	以上
挿入辞	<	小なり
挿入辞	<=	以下
接頭辞	~ (チルド)	ビット単位の NOT
挿入辞	&	ビット単位の AND
挿入辞	|	ビット単位の OR
挿入辞	^	ビット単位の排他的 OR
接頭辞	!	論理否定
挿入辞	&&	論理積
挿入辞	| |	論理和

C では感嘆符(!)は演算子なので，感嘆符をコメント区切り文字として使用することはできません。言語が C に設定されている場合は，感嘆符の代わりに/*がコメント区切り文字として受け入れられます。現在の行の終端までがコメントになります。対応する*/は不要であり，認識もされません。しかし，デバッガのログ・ファイルをデバッガの入力として使用できるようにするため，1 行の中で空白以外の最初の文字が感嘆符(!) である場合は，デバッガはその感嘆符をコメント区切り文字として認識します。

C の言語式とデバッガのアドレス式の両方で，デバッガは接頭辞のアスタリスク(*)を間接参照演算子として受け入れます。言語が C に設定されているときのアドレス式では，接頭辞“*”は接頭辞“.”または“@”と同義です。

デバッガは， C または他のどの言語でも代入演算子は何もサポートしません。これは，デバッグされるプログラムに予期しない変更が加えられるのを防ぐためです。したがって，=， +=，–=， ++，–– などの演算子は認識されません。メモリ記憶位置の内容を変更するには， DEPOSIT コマンドを使用して明示的に変更しなければなりません。

C.6.2 言語式とアドレス式の構造

サポートされている， C の言語式とアドレス式の構造を次に示します。

シンボル	構造
[ ]	添字指定
.(ピリオド)	構造体構成要素の選択
->	ポインタの間接参照

C.6.3 データ型

サポートされている C のデータ型を次に示します。

C のデータ型	VMS のデータ型名
__int64 (Alpha および Integrity 固有)	クォドワード(Q)
unsigned__int64 (Alpha 固有)	クォドワード符号なし(QU)
__int32 (Alpha および Integrity 固有)	ロングワード整数(L)
unsigned__int32 (Alpha および Integrity 固有)	ロングワード符号なし(LU)
int	ロングワード整数(L)
unsigned int	ロングワード符号なし(LU)
__int16 (Alpha および Integrity 固有)	ワード整数(W)
unsigned__int16 (Alpha および Integrity 固有)	ワード符号なし(WU)
short int	ワード整数(W)
unsigned short int	ワード符号なし(WU)
char	バイト整数(B)
unsigned char	バイト符号なし(BU)
float	F 浮動小数点数(F)
__f_float (Alpha および Integrity 固有)	F 浮動小数点数(F)
double	D 浮動小数点数(D)
double	G 浮動小数点数(G)
__g_float (Alpha および Integrity 固有)	G 浮動小数点数(G)
float (Alpha および Integrity 固有)	IEEE S 浮動小数点数(FS)
__s_float (Alpha および Integrity 固有)	IEEE S 浮動小数点数(FS)
double (Alpha および Integrity 固有)	IEEE T 浮動小数点数(FT)
__t_float (Alpha および Integrity 固有)	IEEE T 浮動小数点数(FT)
enum	(なし)
struct	(なし)
union	(なし)
ポインタ型	(なし)
配列型	(なし)

float 型の浮動小数点数は，コンパイラのスイッチによって， F 浮動小数点数または IEEE S 浮動小数点数のどちらかで表現されます。

double 型の浮動小数点数は，コンパイラのスイッチによって， IEEE T 浮動小数点数， D 浮動小数点数，または G 浮動小数点数のどちらかで表現されます。

C.6.4 大文字小文字の区別

C 言語では，シンボル名の大文字と小文字は区別され，それぞれ別個の文字として処理されます。

C.6.5 静的変数と非静的変数

static， globaldef， globalref， extern の各記憶クラスの変数は，静的に割り当てられます。

記憶クラスが auto または register の変数は，非静的に (スタックまたはレジスタに) 割り当てられます。このような非静的変数は，それらを定義しているルーチンが呼び出しスタック上でアクティブになっているときだけアクセス可能です。

C.6.6 スカラ変数

デバッガ・コマンドでは， C のどのデータ型であっても，プログラムのソース・コードに記述するのと全く同じようにスカラ変数を指定することができます。

次の説明では， char 型変数とポインタについて補足します。

デバッガは char 型変数を ASCII 文字ではなく，バイト整数として解釈します。 char 型変数 ch の内容を文字として表示するには，/ASCII 修飾子を使用しなければなりません。

```
DBG> EXAMINE/ASCII ch
SCALARS\main\ch:      "A"
```

また， char 型変数に格納する場合も，/ASCII 修飾子を使用してバイト整数を同値の ASCII 文字に変換する必要があります。次に例を示します。

```
DBG> DEPOSIT/ASCII ch = 'z'
DBG> EXAMINE/ASCII ch
SCALARS\main\ch:        "z"
```

次の宣言と代入が行われている場合に， EXAMINE コマンドでポインタの構文を使用するときの例を示します。

```
static long li  = 790374270;
static int *ptr = &li;

DBG> EXAMINE *ptr
*SCALARS\main\ptr:       790374270
```

C.6.7 配列

デバッガは， C の配列を他のほとんどの言語と同様に処理します。したがって，配列の構文を使用して (たとえば， EXAMINE arr[3]で)，配列の集合体全体，配列断面，または配列の個々の要素をそれぞれチェックすることができます。また，配列への格納は一度に 1 要素ずつです。

C.6.8 文字列

C の文字列は， NULL で終了する ASCII 文字列 (ASCIZ 文字列) として実現されています。文字列全体をチェックする場合，または文字列全体にデータを格納する場合は，デバッガが文字列の終端を正しく解釈できるように，/ASCIZ (または/AZ) 修飾子を使用します。 C の配列の添字を指定する演算子([ ])を使用すると，文字列の個々の文字をチェックしたり，文字列に個々に文字を格納することができます。文字を個々に検査したり格納したりするときは，/ASCII 修飾子を使用してください。

次の宣言と代入が行われているものとします。

```
static char *s = "vaxie";
static char **t = &s;
```

EXAMINE/AZ コマンドは，*s と**t が指す文字列の内容を表示します。

```
DBG> EXAMINE/AZ  *s
*STRING\main\s: "vaxie"
DBG> EXAMINE/AZ  **t
**STRING\main\t:        "vaxie"
```

DEPOSIT/AZ コマンドは，*sが指す変数に新しい ASCIZ 文字列を格納します。文字列の新しい内容を表示するには， EXAMINE/AZ コマンドを使用します。

```
DBG> DEPOSIT/AZ  *s = "DEC C"
DBG> EXAMINE/AZ  *s,  **t
*STRING\main\s: "DEC C"
**STRING\main\t:         "DEC C"
```

配列の添字指定を行うと，文字列の文字を個々にチェックしたり，文字列の特定の記憶位置に新しい ASCII 値を格納することができます。文字列の個々のメンバにアクセスするときは，/ASCII 修飾子を使用してください。それに続けて EXAMINE/AZ コマンドを実行すると，格納された値を含む文字列全体が表示されます。

```
DBG> EXAMINE/ASCII s[3]
[3]:    " "
DBG> DEPOSIT/ASCII s[3] = "-"
DBG> EXAMINE/AZ *s, **t
*STRING\main\s:    "VAX-C"
**STRING\main\t:   "VAX-C"
```

C.6.9 構造体と共用体

構造体をチェックするときは，その全体を検査したり，メンバ単位で検査することができます。また，構造体へのデータの格納は一度に 1 メンバずつ行うことができます。

構造体のメンバまたは共用体のメンバを参照するには， C で参照するときの通常の構文を使用します。つまり，変数pが構造体へのポインタの場合，その構造体のメンバyは，p ->yという式で参照することができます。変数xが，構造体に割り当てられた記憶域のベースを参照している場合，その構造体のメンバはx.yという式で参照することができます。

デバッガは，構造体や共用体のメンバを参照するのに， C の型検査規則を使用します。たとえばx.yの場合，yがxのメンバである必要はありません。yは，型とあわせてオフセットとして処理されます。このような参照があいまいになる場合 (メンバyを持つ構造体が複数ある場合)，デバッガは次の規則に従って参照を解決しようとします。なお，構造体や共用体のメンバ参照のあいまいさを解消するときは，x.yとp ->yの両方とも同じ規則が適用されます。

- メンバの中でyだけが構造体xまたは共用体xに属している場合，yが参照される。
- メンバの中でyだけがxと同じ有効範囲にある場合は，yが参照される。

使用される有効範囲を絞るため，また，あいまいさを解消するために，xを参照するときは，いつでもパス名を与えることができます。パス名は，xとyの両方を検索するのに使用されます。

C.7 C++ バージョン 5.5 および 5.5 以降 (Alpha および Integrity のみ)

Alpha システムおよび Integrity システムでは， OpenVMS デバッガはバージョン 5.5 以降 (Alpha および Integrity のみ) のコンパイラでコンパイルされた C++ モジュールデバッグする機能がサポートされています。

デバッガは， C++ の次の機能をサポートします。

- C++ の名前と式。次のものを含む。
  - クラス・メンバを参照するための明示的な，および暗黙のthisポインタ
  - スコープ解決演算子 (::)
  - メンバ・アクセス演算子であるピリオド (.) と右矢印 (->)
  - テンプレートのインスタンス化
  - テンプレート名
- 次に示す中でのブレークポイントの設定。
  - 静的な仮想関数を含む，メンバ関数

  - オーバーロードされた関数
  - コンストラクタとデストラクタ
  - テンプレートのインスタンス化
  - 演算子
- 関数の呼び出し。オーバーロードされた関数を含む。
- C++ のコードと，他の言語のコードが混在したプログラムのデバッグ。

この節のデバッグの例は， Example C–1 に含まれるテスト・プログラムを参照しており，また， Example C–2 に含まれるデバッグ・セッションの一部を取り出したものになっています。次節以降で， C++ のデバッガ・サポートについて説明します。

C.7.1 言語式における演算子

言語式でサポートされる C++ 演算子を次に示します。

種類	シンボル	機能
接頭辞	*	間接参照
接頭辞	&	アドレス
接頭辞	sizeof	オブジェクトのサイズ
接頭辞	–	単項マイナス (否定)
挿入辞	+	加算
挿入辞	–	減算
挿入辞	*	乗算
挿入辞	/	除算
挿入辞	%	剰余
挿入辞	<<	左シフト
挿入辞	>>	右シフト
挿入辞	==	等しい
挿入辞	!=	等しくない
挿入辞	>	より大きい
挿入辞	>=	以上
挿入辞	<	より小さい
挿入辞	<=	以下
接頭辞	~ (チルド)	ビットごとの NOT
挿入辞	&	ビットごとの AND
挿入辞	|	ビットごとの OR
挿入辞	^	ビットごとの排他的 OR
接頭辞	!	論理 NOT
挿入辞	&&	論理 AND
挿入辞	||	論理 OR

感嘆符(!)は演算子なので， C++ プログラムでコメント区切り文字として使用できません。ただし，デバッガのログ・ファイルをデバッガの入力として使用できるように，行内のスペースでない最初の文字が ! である場合は，デバッガは ! をコメント区切り文字として解釈します。デバッガは C++ 言語モードでは，/*または//をコメントの先頭として解釈し，その行の最後までをコメントとみなします。

デバッガは， C++ 言語式とデバッガ・アドレス式の両方で，アスタリスク(*)接頭辞を間接参照演算子として扱います。デバッガが C++ 言語モードの場合，アドレス式では，*接頭辞はピリオド (.) 接頭辞またはサイン (@) 接頭辞と同じです。

デバッグの対象のプログラムが意図に反して修正されることがないように，デバッガは C++，および他の言語の代入演算子をサポートしません。このため，=， +=，–=，++，–– はデバッガコマンドによって解釈されません。メモリの内容を変更するには，デバッガの DEPOSIT コマンドを使用します。

C.7.2 言語式とアドレス式における構造体

C++ の言語式とアドレス式でサポートされる構造体を次に示します。

シンボル	構造体
[ ]	添字付け
. (ピリオド)	構造体の構成要素の選択
->	ポインタ間接参照
::	スコープ解決

C.7.3 データ型

サポートされる C++ のデータ型を次に示します。

C++ データ型	オペレーティング・システム・データ型名
__int64 (Alpha および Integrity)	クォドワード整数 (Q)
unsigned __int64 (Alpha および Integrity)	クォドワード符号なし整数 (QU)
__int32 (Alpha および Integrity)	ロングワード整数 (L)
unsigned __int32 (Alpha および Integrity)	ロングワード符号なし整数 (LU)
int	ロングワード整数 (L)
unsigned int	ロングワード符号なし整数 (LU)
__int16 (Alpha および Integrity)	ワード整数 (W)
unsigned __int16 (Alpha および Integrity)	ワード符号なし整数 (WU)
short int	ワード整数 (W)
unsigned short int	ワード符号なし整数 (WU)
char	バイト整数 (B)

C++ データ型	オペレーティング・システム・データ型名
unsigned char	バイト符号なし整数 (BU)
float	F 浮動小数点数 (F)
__f_float (Alpha および Integrity)	F 浮動小数点数 (F)
double	D 浮動小数点数 (D)
double	G 浮動小数点数 (G)
__g_float (Alpha および Integrity)	G 浮動小数点数 (G)
float (Alpha および Integrity)	IEEE S 浮動小数点数 (FS)
__s_float (Alpha および Integrity)	IEEE S 浮動小数点数 (FS)
double (Alpha および Integrity)	IEEE T 浮動小数点数 (FT)
__t_float (Alpha および Integrity)	IEEE T 浮動小数点数 (FT)
enum	(なし)
struct	(なし)
class	(なし)
union	(なし)
Pointer Type	(なし)
Array Type	(なし)

float 型の浮動小数点数は，コンパイラのスイッチに応じて， F_Floating または IEEE S_Floating として表現されます。

double 型の浮動小数点数は，コンパイラのスイッチに応じて， IEEE T_Floating, D_Floating，または G_Floating として表現されます。

C.7.4 大文字小文字の区別

C++ では，シンボル名の大文字小文字が区別されます。つまり，大文字と小文字は異なる文字として扱われます。

C.7.5 クラスに関する情報の表示

クラス宣言に関する静的な情報を表示するには， SHOW SYMBOL コマンドを使用します。クラス・オブジェクトに関する動的な情報を表示するには， EXAMINE コマンドを使用します (第 C.7.6 項を参照)。

SHOW SYMBOL/FULL コマンドを使用すると，次の情報を含むクラス型宣言を表示することができます。

データ・メンバ (静的なデータ・メンバを含む)
メンバ関数 (静的なメンバ関数を含む)
コンストラクタとデストラクタ
基底クラスと派生クラス

例を示します。

```
dbg> SHOW SYMBOL /TYPE C
type C
    struct (C, 13 components), size: 40 bytes
overloaded name C
       instance C::C(void)
       instance C::C(const C &)
dbg> SHOW SYMBOL /FULL C
type C
    struct (C, 13 components), size: 40 bytes
      inherits: B1, size: 24 bytes, offset: 0 bytes
                B2, size: 24 bytes, offset: 12 bytes
      contains the following members:
        overloaded name C::g
               instance C::g(int)
               instance C::g(long)
               instance C::g(char)
        j : longword integer, size: 4 bytes, offset: 24 bytes
        s : longword integer, size: 4 bytes, address: #  [static]
        overloaded name C
        int ==(C &)
        C & =(const C &)
        void h(void)   [virtual]
        ~C(void)
        __vptr : typed pointer type, size: 4 bytes, offset: 4 bytes
        __bptr : typed pointer type, size: 4 bytes, offset: 8 bytes
        structure has been padded, size: 4 bytes, offset: 36 bytes
overloaded name C
       instance C::C(void)
       instance C::C(const C &)
DBG>
```

SHOW SYMBOL/FULL コマンドでは，基底クラスまたは派生クラスのメンバが表示されない点に注意してください。これらのクラスのメンバについては， SHOW SYMBOL/FULL base_class_name，および SHOW SYMBOL/FULL derived_class_name コマンドを使用します。例を示します。

```
DBG> SHOW SYMBOL /FULL B1
type B1
    struct (B1, 8 components), size: 24 bytes
      inherits: virtual A
      is inherited by: C
      contains the following members:
        i : longword integer, size: 4 bytes, offset: 0 bytes
        overloaded name B1
        void f(void)
        B1 & =(const B1 &)
        void h(void)   [virtual]
        __vptr : typed pointer type, size: 4 bytes, offset: 4 bytes
        __bptr : typed pointer type, size: 4 bytes, offset: 8 bytes
        structure has been padded, size: 12 bytes, offset: 12 bytes
overloaded name B1
       instance B1::B1(void)
       instance B1::B1(const B1 &)
DBG>
```

クラス・メンバに関する情報を表示するには， SHOW SYMBOL/FULL class_member_name コマンドを使用します。例を示します。

```
DBG> SHOW SYMBOL /FULL j
record component C::j
    address: offset 24 bytes from beginning of record
    atomic type, longword integer, size: 4 bytes
record component A::j
    address: offset 4 bytes from beginning of record
    atomic type, longword integer, size: 4 bytes
DBG>
```

オブジェクトに関する詳細情報を表示するには， SHOW SYMBOL/FULL コマンドを使用します。

現在， SHOW SYMBOL コマンドは修飾された名前をサポートしていない点に注意してください。たとえば，次に示すコマンドは現在サポートされていません。

```
SHOW SYMBOL    object_name.function_name
SHOW SYMBOL    class_name::member_name
```

C.7.6 オブジェクトに関する情報の表示

デバッガはオブジェクトに関する情報を表示するために， C++ のシンボル検索規則を使用します。現在のオブジェクトの値を表示するには， EXAMINE コマンドを使用します。例を示します。

```
DBG> EXAMINE a
CXXDOCEXAMPLE\main\a: struct A
    i:  0
    j:  1
    __vptr:     131168
DBG>
```

また，EXAMINE コマンドでメンバ・アクセス演算子であるピリオド (.) と右矢印 (->) を使用することによって，個々のオブジェクト・メンバを表示することもできます。例を示します。

```
DBG> EXAMINE ptr
CXXDOCEXAMPLE\main\ptr:         40
DBG> EXAMINE *ptr
*CXXDOCEXAMPLE\main\ptr: struct A
    i:  0
    j:  1
    __vptr:     131168
DBG> EXAMINE a.i
CXXDOCEXAMPLE\main\a.i: 0
DBG> EXAMINE ptr->i
CXXDOCEXAMPLE\main\ptr->i:      0
DBG>
```

デバッガは，仮想的な継承を正しく解釈します。例を示します。

```
DBG> EXAMINE c
CXXDOCEXAMPLE\main\c: struct C
    inherit B1
        inherit virtual A
            i:  8
            j:  9
            __vptr:     131200
        i:      10
        __vptr: 131232
        __bptr: 131104
    inherit B2
        inherit virtual A  (already printed, see above)
        i:      11
        __vptr: 131280
        __bptr: 131152
    j:  12
    __vptr:     131232
    __bptr:     131104
DBG>
```

グローバル変数を参照したり，基底クラスの隠蔽メンバを参照したり，継承されたメンバを明示的に参照したり，現在のコンテキストによって隠蔽されたメンバを命名したりするために，スコープ解決演算子 (::) を使用します。例を示します。

```
DBG> EXAMINE c.j
CXXDOCEXAMPLE\main\c.j: 12
DBG> EXAMINE c.A::j
CXXDOCEXAMPLE\main\c.A::j:      9
DBG> EXAMINE x
CXXDOCEXAMPLE\main\x:   101
DBG> EXAMINE ::x
CXXDOCEXAMPLE\x:        13
DBG>
```

デバッガはあいまいなメンバ参照を解決するために，参照を満たすメンバをリスト表示して，メンバに対するあいまいでない参照の指定を求めます。例を示します。

```
DBG> EXAMINE c.i
%DEBUG-I-AMBIGUOUS, 'i' is ambiguous, matching the following
    CXXDOCEXAMPLE\main\c.B1::i
    CXXDOCEXAMPLE\main\c.B2::i
%DEBUG-E-REENTER, reenter the command using a more precise pathname
DBG> EXAMINE c.B1::i
CXXDOCEXAMPLE\main\c.B1::i:     10
DBG>
```

静的なデータ・メンバを参照するには，スコープ解決演算子 (::) を使用します。例を示します。

```
DBG> EXAMINE c.s
CXXDOCEXAMPLE\main\c.s: 42
DBG> EXAMINE C::s
C::s:   42
DBG>
```

オブジェクトのクラス型を表示するには， SHOW SYMBOL/FULL コマンドを使用します (第 C.7.5 項を参照)。

C.7.7 ウォッチポイントの設定

オブジェクトのウォッチポイントを設定することができます。静的でないすべてのデータ・メンバはウォッチされます (基底クラス中のデータ・メンバを含みます)。オブジェクト上にウォッチポイントを設定すると，静的なデータ・メンバはウォッチされません。ただし，静的なデータ・メンバについて，明示的にウォッチポイントを設定することは可能です。例を示します。

```
DBG> SET WATCH c
%DEBUG-I-WPTTRACE, non-static watchpoint, tracing every instruction
DBG> GO
watch of CXXDOCEXAMPLE\main\c.i at CXXDOCEXAMPLE\main\%LINE 50+8
    50:     c.B2::i++;
   old value: 11
   new value: 12
break at CXXDOCEXAMPLE\main\%LINE 51
    51:     c.s++;
DBG> SET WATCH c.s
DBG> GO
watch of CXXDOCEXAMPLE\main\c.s at CXXDOCEXAMPLE\main\%LINE 51+16
    51:     c.s++;
   old value: 43
   new value: 44
break at CXXDOCEXAMPLE\main\%LINE 53
    53:     b1.f();
DBG>
```

C.7.8 デバッグ関数

デバッガはメンバ関数の情報を表示するために，C++ シンボル検索ルールを使用します。例を示します。

```
DBG> EXAMINE /SOURCE b1.f
module CXXDOCEXAMPLE
    14:     void f() {}
DBG> SET BREAK B1::f
DBG> GO
break at routine B1::f
    14:     void f() {}
DBG>
```

デバッガは，thisポインタに対する参照を正しく解釈します。例を示します。

```
DBG> EXAMINE this
B1::f::this:           16
DBG> EXAMINE *this
*B1::f::this: struct B1
    inherit virtual A
        i:      2
        j:      3
        __vptr: 131184
    i:  4
    __vptr:     131248
    __bptr:     131120
DBG> EXAMINE this->i
B1::f::this->i: 4
DBG> EXAMINE this->j
B1::f::this->A::j:      3
DBG>EXAMINE i
B1::f::this->i: 4
DBG> EXAMINE j
B1::f::this->A::j:      3
DBG>
```

デバッガは，仮想メンバ関数を正しく参照します。例を示します。

```
DBG> EXAMINE /SOURCE %LINE 53
module CXXDOCEXAMPLE
    53:     b1.f();
DBG> SET BREAK this->h
DBG> SHOW BREAK
breakpoint at routine B1::f
breakpoint at routine B1::h
!!
!! We are at the call to B1::f made at 'c.B1::f()'.
!! Here this->h matches C::h.
!!
DBG> GO
break at routine B1::f
    14:     void f() {}
DBG> EXAMINE /SOURCE %LINE 54
module CXXDOCEXAMPLE
    54:     c.B1::f();
DBG> SET BREAK this->h
DBG> SHOW BREAK
breakpoint at routine B1::f
breakpoint at routine B1::h
breakpoint at routine C::h
```

```
!!
!! Handling overloaded functions
!!
DBG> show symbol/full g
overloaded name C::g
routine C::g(char)
type signature: void g(char)
address: 132224, size: 128 bytes
routine C::g(long)
type signature: void g(long)
address: 132480, size: 96 bytes
DBG> SET BREAK g
%DEBUG-I-NOTUNQOVR, symbol 'g' is overloaded
overloaded name C::g
       instance C::g(int)
       instance C::g(long)
       instance C::g(char)
%DEBUG-E-REENTER, reenter the command using a more precise pathname
DBG> SET BREAK g(int)

DBG> CANCEL BREAK/ALL
DBG>
```

オーバーロードされた関数上にブレークを設定すると，デバッガは関数のインスタンスをリスト表示し，コンパイラが生成した固有のラベルを使って，正しいインスタンスを指定するよう求めます。デバッガ・バージョン 7.2 を使った例を示します。

```
DBG> SET BREAK g
%DEBUG-I-NOTUNQOVR, symbol 'g' is overloaded
overloaded name C::g
       instance void g(int)
       instance void g(long)
       instance void g(char *)
%DEBUG-E-REENTER, reenter the command using a more precise pathname
DBG> SET BREAK g(int)
DBG>
```

注意

オーバーロードされた関数についての表示方法と指定方法は， OpenVMS **デバッガ・バージョン** 7.1C **とは異なります。デバッガ・バージョン** 7.1C **を使った例を示します。**

デバッガは，コンストラクタ，デストラクタ，および演算子のデバッグをサポートします。例を示します。

```
DBG> SET BREAK C
%DEBUG-I-NOTUNQOVR, symbol 'C' is overloaded
overloaded name C
       instance C::C(void)
       instance C::C(const C &)
%DEBUG-E-REENTER, reenter the command using a more precise pathname
DBG> SHOW SYMBOL /FULL ~C
routine C::~C
   type signature: ~C(void)
    code address: #, size: 152 bytes
    procedure descriptor address: #
DBG> SET BREAK ~C
DBG> SET BREAK %NAME'=='
%DEBUG-W-UNALLOCATED, '==' is not allocated in memory (optimized away)
%DEBUG-E-CMDFAILED, the SET BREAK command has failed
DBG> SHOW SYMBOL /FULL ==,
routine c::operator==, type
signature: bool operator==M-)
code address: 198716, size:
40 bytes, procedure descriptor
address: 65752
DBG> SET BREAK operator==
DBG> SHOW SYMBOL /FULL ==
routine C::==
    type signature: int ==(C &)
    address: unallocated
DBG> SHOW BREAK
breakpoint at routine C::~C
DBG>
DBG> examine C::~C
C::~C: alloc r35 = ar.pfs, 3F, 01, 00
DBG>
DBG> ex/source ~C
module CXXDOCEXAMPLE
37: ~C() {}
```

C.7.9 C++ デバッガ・サポートに関する制限事項

C++ プログラムをデバッグする場合は，次の制限事項が適用されます。

- デバッガ・コマンドでは，テンプレートを名前で指定することは できません。テンプレートのインスタンスの名前を指定しなけれ ばなりません。
- C++ では，インスタンス化されたテンプレート名の中の式を，stack<double,f*10>のような完全な定数式とすることができます。 この形式は，デバッガではまだサポートされていません。 このため，式の値を入力することが必要です (たとえば， stack の 例で f が 10 であれば， 100 を入力しなければなりません)。

Example C–1 C++ プログラム例，CXXDOCEXAMPLE.C

```
int x = 0;

struct A
    {
    int i,j;
    void f() {}
    virtual void h() {};
    A() { i=x++; j=x++; }
    };
struct B1 : virtual A
    {
    int i;
    void f() {}
    virtual void h() {}
    B1() { i=x++; }
    };
struct B2 : virtual A
    {
    int i;
    void f() {}
    virtual void h() {}
    B2() { i=x++; }
    };
struct C : B1, B2
    {
    int j;
    static int s;
    void g( int ) {}
    void g( long ) {}
    void g( char * ) {}
    void h() {}
    operator ==( C& ) { return 0; }
    C() { j=x++; }
    ~C() {}
    };

int C::s = 42;
main()
    {
    A a; B1 b1; B2 b2; C c;
    A *ptr = &a;
    int x = 101;
    x++;

    c.s++;
    c.B2::i++;
    c.s++;

    b1.f();
    c.B1::f();
    }
```

Example C–2 は Example C–1 に含まれているプログラムのデバッグ・セッションの例です。

Example C–2 C++ デバッグ例

```
DBG> GO
break at routine CXXDOCEXAMPLE\main
    44:     A a; B1 b1; B2 b2; C c;
DBG> STEP
stepped to CXXDOCEXAMPLE\main\%LINE 45
    45:     A *ptr = &a;
DBG> STEP
stepped to CXXDOCEXAMPLE\main\%LINE 46
    46:     int x = 101;
DBG> STEP
stepped to CXXDOCEXAMPLE\main\%LINE 47
    47:     x++;
!!
!! Displaying class information
!!
DBG> SHOW SYMBOL /TYPE C
type C
    struct (C, 13 components), size: 40 bytes
overloaded name C
       instance C::C(void)
       instance C::C(const C &)
DBG> SHOW SYMBOL /FULL C
type C
    struct (C, 13 components), size: 40 bytes
      inherits: B1, size: 24 bytes, offset: 0 bytes
                B2, size: 24 bytes, offset: 12 bytes
      contains the following members:
        overloaded name C::g
               instance C::g(int)
               instance C::g(long)
               instance C::g(char)
        j : longword integer, size: 4 bytes, offset: 24 bytes
        s : longword integer, size: 4 bytes, address: #  [static]
        overloaded name C
        int ==(C &)
        C & =(const C &)
        void h(void)   [virtual]
        ~C(void)
        __vptr : typed pointer type, size: 4 bytes, offset: 4 bytes
        __bptr : typed pointer type, size: 4 bytes, offset: 8 bytes
        structure has been padded, size: 4 bytes, offset: 36 bytes
overloaded name C
       instance C::C(void)
       instance C::C(const C &)
```

(次ページに続く)

Example C–2 (続き) C++ デバッグ例

```
!!
!! Displaying information about base classes
!!
DBG> SHOW SYMBOL /FULL B1
type B1
    struct (B1, 8 components), size: 24 bytes
      inherits: virtual A
      is inherited by: C
      contains the following members:
        i : longword integer, size: 4 bytes, offset: 0 bytes
        overloaded name B1
        void f(void)
        B1 & =(const B1 &)
        void h(void)   [virtual]
        __vptr : typed pointer type, size: 4 bytes, offset: 4 bytes
        __bptr : typed pointer type, size: 4 bytes, offset: 8 bytes
        structure has been padded, size: 12 bytes, offset: 12 bytes
overloaded name B1
       instance B1::B1(void)
       instance B1::B1(const B1 &)
!!
!! Displaying class member information
!!
DBG> SHOW SYMBOL /FULL j
record component C::j
    address: offset 24 bytes from beginning of record
    atomic type, longword integer, size: 4 bytes
record component A::j
    address: offset 4 bytes from beginning of record
    atomic type, longword integer, size: 4 bytes
```

(次ページに続く)

Example C–2 (続き) C++ デバッグ例

```
!!
!! Simple object display
!!
DBG> EXAMINE a
CXXDOCEXAMPLE\main\a: struct A
    i:  0
    j:  1
    __vptr:     131168
!!
!! Using *, -> and . to access objects and members
!!
DBG> EXAMINE ptr
CXXDOCEXAMPLE\main\ptr:         40
DBG> EXAMINE *ptr
*CXXDOCEXAMPLE\main\ptr: struct A
    i:  0
    j:  1
    __vptr:     131168
DBG> EXAMINE a.i
CXXDOCEXAMPLE\main\a.i: 0
DBG> EXAMINE ptr->i
CXXDOCEXAMPLE\main\ptr->i:      0
!!
!! Complicated object example
!!
DBG> EXAMINE c
CXXDOCEXAMPLE\main\c: struct C
    inherit B1
        inherit virtual A
            i:  8
            j:  9
            __vptr:     131200
        i:      10
        __vptr: 131232
        __bptr: 131104
    inherit B2
        inherit virtual A  (already printed, see above)
        i:      11
        __vptr: 131280
        __bptr: 131152
    j:  12
    __vptr:     131232
    __bptr:     131104
!!
!! The debugger using C++ symbol lookup rules (to match c.j)
!! and then the use of :: to specify a particular member named j.
!!
DBG> EXAMINE c.j
CXXDOCEXAMPLE\main\c.j: 12
DBG> EXAMINE c.A::j
CXXDOCEXAMPLE\main\c.A::j:      9
```

(次ページに続く)

Example C–2 (続き) C++ デバッグ例

```
!!
!! Using the global scope resolution operator.
!!
DBG> EXAMINE x
CXXDOCEXAMPLE\main\x:   101
DBG> EXAMINE ::x
CXXDOCEXAMPLE\x:        13
!!
!! Handling ambiguous member references.
!!
DBG> EXAMINE c.i
%DEBUG-I-AMBIGUOUS, 'i' is ambiguous, matching the following
    CXXDOCEXAMPLE\main\c.B1::i
    CXXDOCEXAMPLE\main\c.B2::i
%DEBUG-E-REENTER, reenter the command using a more precise pathname
DBG> EXAMINE c.B1::i
CXXDOCEXAMPLE\main\c.B1::i:     10
!!
!! Refering to static data members: with . and with ::
!!
DBG> EXAMINE c.s
CXXDOCEXAMPLE\main\c.s: 42
DBG> EXAMINE C::s
C::s:   42
!!
!! Setting watchpoints on objects.  All non-static data members
!! are watched (including those in base classes).  Static data
!! members are not watched.  Of course watchpoints on static data
!! members can be set explicitly.
!!
DBG> SET WATCH c
%DEBUG-I-WPTTRACE, non-static watchpoint, tracing every instruction
DBG> GO
watch of CXXDOCEXAMPLE\main\c.i at CXXDOCEXAMPLE\main\%LINE 50+8
    50:     c.B2::i++;
   old value: 11
   new value: 12
break at CXXDOCEXAMPLE\main\%LINE 51
    51:     c.s++;
DBG> SET WATCH c.s
DBG> GO
watch of CXXDOCEXAMPLE\main\c.s at CXXDOCEXAMPLE\main\%LINE 51+16
    51:     c.s++;
   old value: 43
   new value: 44
break at CXXDOCEXAMPLE\main\%LINE 53
    53:     b1.f();
```

(次ページに続く)

Example C–2 (続き) C++ デバッグ例

```
!!
!! Basic member lookup applies to functions.
!!
DBG> EXAMINE /SOURCE b1.f
module CXXDOCEXAMPLE
    14:     void f() {}
DBG> SET BREAK B1::f
DBG> GO
break at routine B1::f
    14:     void f() {}
!!
!! Support for 'this'.
!!
DBG> EXAMINE this
B1::f::this:            16
DBG> EXAMINE *this
*B1::f::this: struct B1
    inherit virtual A
        i:      2
        j:      3
        __vptr: 131184
    i:  4
    __vptr:     131248
    __bptr:     131120
DBG> EXAMINE this->i
B1::f::this->i: 4
DBG> EXAMINE this->j
B1::f::this->A::j:      3
DBG>EXAMINE i
B1::f::this->i: 4
DBG> EXAMINE j
B1::f::this->A::j:      3
!!
!! Support for virtual functions.
!!
!! We are at the call to B1::f made at 'b1.f()'.
!! Here this->h matches B1::h.
!!
DBG> EXAMINE /SOURCE %LINE 53
module CXXDOCEXAMPLE
    53:     b1.f();
DBG> SET BREAK this->h
DBG> SHOW BREAK
breakpoint at routine B1::f
breakpoint at routine B1::h
```

(次ページに続く)

Example C–2 (続き) C++ デバッグ例

```
!!
!! We are at the call to B1::f made at 'c.B1::f()'.
!! Here this->h matches C::h.
!!
DBG> GO
break at routine B1::f
    14:     void f() {}
DBG> EXAMINE /SOURCE %LINE 54
module CXXDOCEXAMPLE
    54:     c.B1::f();
DBG> SET BREAK this->h
DBG> SHOW BREAK
breakpoint at routine B1::f
breakpoint at routine B1::h
breakpoint at routine C::h
!!
!! Handling overloaded functions
!!
DBG> SET BREAK g
%DEBUG-I-NOTUNQOVR, symbol 'g' is overloaded
overloaded name C::g
       instance C::g(int)
       instance C::g(long)
       instance C::g(char)
%DEBUG-E-REENTER, reenter the command using a more precise pathname
DBG> SET BREAK g(int)

DBG> CANCEL BREAK/ALL
!!
!! Working with constructors, destructors, and operators.
!!
DBG> SET BREAK C
%DEBUG-I-NOTUNQOVR, symbol 'C' is overloaded
overloaded name C
       instance C::C(void)
       instance C::C(const C &)
%DEBUG-E-REENTER, reenter the command using a more precise pathname
DBG> SHOW SYMBOL /FULL ~C
routine C::~C
   type signature: ~C(void)
    code address: #, size: 152 bytes
    procedure descriptor address: #
DBG> SET BREAK %NAME'~C'
DBG> SET BREAK %NAME'=='
%DEBUG-W-UNALLOCATED, '==' is not allocated in memory (optimized away)
%DEBUG-E-CMDFAILED, the SET BREAK command has failed
DBG> SHOW SYMBOL /FULL ==
routine C::==
    type signature: int ==(C &)
    address: unallocated
DBG> SHOW BREAK
```

(次ページに続く)

Example C–2 (続き) C++ デバッグ例

```
breakpoint at routine C::~C
DBG> EXIT
```

C.8 COBOL

次の各サブトピックでは，デバッガによる COBOL のサポートについて説明します。

C.8.1 言語式の演算子

言語式でサポートされている COBOL の演算子を次に示します。

種類	シンボル	機能
接頭辞	+	単項正符号
接頭辞	–	単項負符号 (否定)
挿入辞	+	加算
挿入辞	–	減算
挿入辞	*	乗算
挿入辞	/	除算
挿入辞	=	等値
挿入辞	NOT =	不等
挿入辞	>	大なり
挿入辞	NOT <	以上
挿入辞	<	小なり
挿入辞	NOT >	以下
挿入辞	NOT	論理否定
挿入辞	AND	論理積
挿入辞	OR	論理和

C.8.2 言語式とアドレス式の構造

サポートされている COBOL の言語式とアドレス式の構造を次に示します。

シンボル	構造
( )	添字指定
OF	レコードの構成要素の選択
IN	レコードの構成要素の選択

C.8.3 データ型

サポートされている COBOL のデータ型を次に示します。

COBOL のデータ型	VMS のデータ型名
COMP	ロングワード整数 (L,LU)
COMP	ワード整数 (W,WU)
COMP	クォドワード整数 (Q,QU)
COMP-1	F 浮動小数点数(F)
COMP-1 (Alpha および Integrity 固有)	IEEE S 浮動小数点数(FS)
COMP-2	D 浮動小数点数(D)
COMP-2 (Alpha および Integrity 固有)	IEEE T 浮動小数点数(FT)
COMP-3	パック 10 進数(P)
INDEX	ロングワード整数(L)
英数字	ASCII テキスト(T)
レコード	(なし)
数値符号なし	数値文字列，符号なし (NU)
先行分離記号	数値文字列，左分離記号 (NL)
先行オーバパンチ記号	数値文字列，左オーバパンチされた記号 (NLO)
終了分離記号	数値文字列, 右分離記号 (NR)
後続オーバパンチ記号	数値文字列，右オーバパンチされた記号 (NRO)

COMP-1 型の浮動小数点数は，コンパイラのスイッチによって， F 浮動小数点数または IEEE S 浮動小数点数のどちらかで表現されます。

COMP-2 型の浮動小数点数は，コンパイラのスイッチによって， D 浮動小数点数または IE EE T 浮動小数点数のどちらかで表現されます。

C.8.4 ソース表示

デバッガは， COPY 文または COPY REPLACING 文でプログラムに取り込まれたソース・テキストを表示することができます。しかし， COPY REPLACING 文または REPLACE 文を使用した場合は， COPY REPLACING 文または REPLACE 文で作成された修正後のソース・テキストではなく，元のソース・テキストが表示されます。

デバッガは， REPORT 節のコードに対応する元のソース行を表示することはできません。 REPORT に対応する DATA SECTION のソース行を見ることはできますが，レポートを作成するためのコンパイル済みコードに対応するソース行は表示されません。

C.8.5 COBOL の INITIALIZE 文と大きいテーブル (配列) (Alpha のみ)

OpenVMS Alpha システムでは，大きいテーブル (配列) が初期化されるときに，STEP コマンドを使用して COBOL プログラムで INITIALIZE 文を実行する場合，デバッガは非常に長い時間と多くのリソースを使用することがあります。

この問題を回避するには， INITIALIZE 文をステップ実行するのではなく，INITIALIZE 文の後の最初の実行可能な行にブレークポイントを設定します。

C.9 DIBOL (VAX のみ)

次の各サブトピックでは，デバッガによる DIBOL のサポートについて説明します。

C.9.1 言語式の演算子

言語式でサポートされている DIBOL の演算子を次に示します。

種類	シンボル	機能
接頭辞	#	丸め
接頭辞	+	単項正符号
接頭辞	–	単項負符号 (否定)
挿入辞	+	加算
挿入辞	–	減算
挿入辞	*	乗算
挿入辞	/	除算
挿入辞	//	商が小数となる除算
挿入辞	.EQ.	等値
挿入辞	.NE.	不等
挿入辞	.GT.	大なり
挿入辞	.GE.	以上
挿入辞	.LT.	小なり
挿入辞	.LE.	以下
挿入辞	.NOT.	論理否定
挿入辞	.AND.	論理積
挿入辞	.OR.	論理和
挿入辞	.XOR.	排他的論理和

C.9.2 言語式とアドレス式の構造

サポートされている DIBOL の言語式とアドレス式の構造を次に示します。

シンボル	構造
( )	部分文字列
[ ]	添字指定
.(ピリオド)	レコードの構成要素の選択

C.9.3 データ型

サポートされている DIBOL のデータ型を次に示します。

DIBOL のデータ型	VMS のデータ型名
I1	バイト整数(B)
I2	ワード整数(W)
I4	ロングワード整数(L)
Pn	パック 10 進数文字列(P)
Pn.m	パック 10 進数文字列(P)
Dn	数値文字列，ゾーン記号 (NZ)
Dn.m	数値文字列，ゾーン記号 (NZ)
An	ASCII テキスト(T)
配列	(なし)
レコード	(なし)

C.10 Fortran

次の各サブトピックでは，デバッガによる Fortran のサポートについて説明します。

C.10.1 言語式の演算子

言語式でサポートされている Fortran の演算子を次に示します。

種類	シンボル	機能
接頭辞	+	単項正符号
接頭辞	–	単項負符号 (否定)
挿入辞	+	加算
挿入辞	–	減算
挿入辞	*	乗算
挿入辞	/	除算
挿入辞	//	連結
挿入辞	.EQ.	等値
挿入辞	= =	等値
挿入辞	.NE.	不等
挿入辞	/=	不等
挿入辞	.GT.	大なり
挿入辞	>	大なり
挿入辞	.GE.	以上
挿入辞	>=	大なりまたは等値
挿入辞	.LT.	小なり
挿入辞	<	小なり
挿入辞	.LE.	以下
挿入辞	<=	小なりまたは等値
接頭辞	.NOT.	論理否定
挿入辞	.AND.	論理積
挿入辞	.OR.	論理和
挿入辞	.XOR.	排他的論理和
挿入辞	.EQV.	同値
挿入辞	.NEQV.	排他的論理和

C.10.2 言語式とアドレス式の構造

サポートされている Fortran の言語式とアドレス式の構造を次に示します。

シンボル	構造
( )	添字指定

シンボル	構造
.(ピリオド)	レコードの構成要素の選択
% (パーセント記号)	レコードの構成要素の選択

C.10.3 定義済みのシンボル

サポートされている Fortran の定義済みのシンボルを次に示します。

シンボル	説明
.TRUE.	論理値 TRUE
.FALSE.	論理値 FALSE

C.10.4 データ型

サポートされている Fortran のデータ型を次に示します。

Fortran のデータ型	VMS のデータ型名
LOGICAL*1	バイト符号なし (BU)
LOGICAL*2	ワード符号なし (WU)
LOGICAL*4	ロングワード符号なし (LU)
LOGICAL*8 (Alpha および Integrity 固有)	クォドワード符号なし (QU)
BYTE	バイト(B)
INTEGER*1	バイト整数(B)
INTEGER*2	ワード整数(W)
INTEGER*4	ロングワード整数(L)
INTEGER*8 (Alpha および Integrity 固有)	クォドワード整数(Q)
REAL*4	F 浮動小数点数(F)
REAL*4 (Alpha および Integrity 固有)	IEEE S 浮動小数点数(FS)
REAL*8	D 浮動小数点数(D)
REAL*8	G 浮動小数点数(G)
REAL*8 (Alpha および Integrity 固有)	IEEE T 浮動小数点数(FT)
REAL*16 (Alpha および Integrity 固有)	H 浮動小数点数(H)
COMPLEX*8	F 複素数 (FC)
COMPLEX*8 (Alpha および Integrity 固有)	IEEE S 浮動小数点数 (SC)
COMPLEX*16	D 複素数 (DC)
COMPLEX*16	G 複素数 (GC)

Fortran のデータ型	VMS のデータ型名
COMPLEX*16 (Alpha および Integrity 固有)	IEEE T 浮動小数点数 (TC)(Alpha 固有)
CHARACTER	ASCII テキスト(T)
配列	(なし)
レコード	(なし)

LOGICAL データ型を表すのに内部的には VMS の符号なし整数のデータ型 (BU, WU, LU, QU) が使用されるとしても，LOGICAL 変数および LOGICAL の値が言語式で使用されている場合は，デバッガはコンパイラと同様に LOGICAL の変数および値を符号付きとして処理します。

デバッガは LOGICAL 変数および LOGICAL 式の値を.TRUE. や.FALSE. ではなく，数値でプリントします。LOGICAL の変数や値で有効なのは，通常は下位ビットだけで，0 が.FALSE.，1 が.TRUE. です。しかし，Fortran では，LOGICAL の値のすべてのビットを操作することが許されており，言語式の中で LOGICAL の値を使用することができます。この理由から，LOGICAL 変数や LOGICAL 式の整数値全体を見なければならない場合があるので，デバッガがそれを表示します。

(1.0,2.0) などの COMPLEX 定数は，デバッガの式ではサポートしていません。

REAL*4 型および COMPLEX*8 型の浮動小数点数は，コンパイラのスイッチに応じて，F 浮動小数点数または IEEE S 浮動小数点数で表現されることがあります。

REAL*8 型および COMPLEX*16 型の浮動小数点数は，コンパイラのスイッチに応じて，D 浮動小数点数，G 浮動小数点数，または IEEE T 浮動小数点数で表現されることがあります。

OpenVMS Alpha システムでは，デバッガは複素数変数を含む式を評価できません。この問題を回避するには，複素数変数の内容を調べ，EXAMINE コマンドによって表示された複素数変数の実数部と虚数部を使用して，式を評価します。

C.10.5 初期化コード

/CHECK=UNDERFLOW 修飾子または/PARALLEL 修飾子を使用してコンパイルしたプログラムをデバッグする場合には，次の例に示すようなメッセージが表示されます。

```
DBG> RUN FORMS
Language: FORTRAN, Module: FORMS
Type GO to reach main program
DBG>
```

“Type GO to reach MAIN program”というメッセージは，メイン・プログラムを開始する前に実行が中断されるため，デバッガの制御のもとで初期化コードを実行できることを示します。 GO コマンドを入力すると，メイン・プログラムの先頭に設定されます。その時点で， GO コマンドを入力すると，他の Fortran プログラムの場合と同様に，プログラムの実行を開始できます。

C.11 MACRO–32

次の節では，デバッガによる MACRO–32 のサポートについて説明します。

C.11.1 言語式の演算子

MACRO–32 言語には，高級言語の式と同じ意味での式というものはありません。受け入れられるのはアセンブリ時の式と，限られた数の演算子だけです。デバッグ時に MACRO–32 のプログラミングで，他の言語の場合と同じくらい自由に式を使用できるようにするため，デバッガは， MACRO–32 自体には含まれていない多数の演算子を MACRO–32 の言語式の中で受け入れるようになっています。特に， BLISS 以後にモデル化された比較演算子とブール演算子はすべて受け入れます。間接参照演算子と通常の算術演算子も受け入れられます。

種類	シンボル	機能
接頭辞	@	間接参照
接頭辞	.	間接参照
接頭辞	+	単項正符号
接頭辞	–	単項負符号 (否定)
挿入辞	+	加算
挿入辞	–	減算
挿入辞	*	乗算
挿入辞	/	除算
挿入辞	MOD	剰余
挿入辞	@	左シフト
挿入辞	EQL	等値
挿入辞	EQLU	等値
挿入辞	NEQ	不等
挿入辞	NEQU	不等
挿入辞	GTR	大なり
挿入辞	GTRU	大なり符号なし
挿入辞	GEQ	以上
挿入辞	GEQU	以上符号なし
挿入辞	LSS	小なり
挿入辞	LSSU	小なり符号なし

種類	シンボル	機能
挿入辞	LEQ	以下
挿入辞	LEQU	以下符号なし
接頭辞	NOT	ビット単位の NOT
挿入辞	AND	ビット単位の AND
挿入辞	OR	ビット単位の OR
挿入辞	XOR	ビット単位の排他的論理和
挿入辞	EQV	ビット単位の同値

C.11.2 言語式とアドレス式の構造

サポートされている， MACRO–32 の言語式とアドレス式の構造を次に示します。

シンボル	構造
[ ]	添字指定
<p,s,e>	BLISS 同様のビットフィールドの選択

MACRO–32 アセンブラが作成する DST 情報では，記憶域を割り当てるアセンブラ指示文の前にあるラベルは，そのラベルが変数名である配列変数として扱われます。その結果，このようなデータを検査または操作するときに，高級言語の配列の構文を使用することができます。

次の MACRO–32 ソース・コードの例では， 4 個のワードに格納されている 16 進数のデータを LAB4 ラベルで指定しています。

```
LAB4:    .WORD    ^X3F,5[2],^X3C
```

デバッガは LAB4 を配列変数として処理します。たとえば次のコマンドは，各要素 (ワード) に格納されている値を表示します。

```
DBG> EXAMINE LAB4
.MAIN.\MAIN\LAB4
    [0]:        003F
    [1]:        0005
    [2]:        0005
    [3]:        003C
```

次のコマンドは， 4 番目のワードに格納されている値を表示します。 1 番目のワードは要素“0”で表されます。

```
DBG> EXAMINE LAB4[3]
.MAIN.\MAIN\LAB4[3]:    03C
```

C.11.3 データ型

MACRO–32 は，データ型をラベル名にバイトします。バインドは，ラベル定義に続くアセンブラ指示文に従って行なわれます。サポートされている MACRO–32 の指示文を次に示します。

MACRO–32 の指示文	VMS のデータ型名
.BYTE	バイト符号なし (BU)
.WORD	ワード符号なし (WU)
.LONG	ロングワード符号なし (LU)
.SIGNED_BYTE	バイト整数(B)
.SIGNED_WORD	ワード整数(W)
.LONG	ロングワード整数(L)
.QUAD	クォドワード整数(Q)
.F_FLOATING	F 浮動小数点数(F)
.D_FLOATING	D 浮動小数点数(D)
.G_FLOATING	G 浮動小数点数(G)
(該当なし)	パック 10 進数(P)

C.11.4 MACRO–32 コンパイラ (AMACRO - Alpha 専用， IMACRO - Integrity 専用)

MACRO–32 で記述したアプリケーションを Alpha システムに移植する場合は，MACRO–32 コンパイラ (AMACRO) を使用します。 MACRO–32 でコンパイルされたコードをデバッグするためのデバッグ・セッションは，アセンブルされたコードの場合に似ていますが，本項で説明するような重要な相違もあります。これらのアプリケーションを移植する場合の説明については，『Porting VAX MACRO Code from OpenVMS VAX to OpenVMS Alpha』を参照してください。

C.11.4.1 コードの再配置

大きな相違は，コードがコンパイルされているという事実です。 VAX システムでは， MACRO–32 の各命令は単一のマシン命令です。 Alpha システムでは，MACRO–32 の各命令は， Alpha の複数のマシン命令へとコンパイルされることがあります。この相違によって大きな副作用が生じます。それは，コンパイル・コマンドで/NOOPTIMIZE を指定しない場合，コードの再配置と再スケジューリングが行われるということです。コードのデバッグが終わってから，/NOOPTIMIZE を指定しないでリコンパイルすることにより性能を改善できます。

C.11.4.2 シンボリック変数

コンパイルされたコードをデバッグする場合とアセンブルされたコードをデバッグする場合のもう一つの大きな違いは， MACRO-32 の新しいコンセプトであるルーチン引数を調べるためのシンボリック変数の定義です。この引数は， Alpha および Integrity サーバのメモリのベクター内には存在しません。

コンパイルしたコードでは，引数は，以下の項目を組み合せたものとして存在することができます。

- レジスタ
- ルーチンのスタック・フレームの上のスタック
- スタック・フレーム (引数リストがホーム・ポジションにあった場合，またはレジスタ引数の保管を伴うルーチンからの呼び出しがあった場合)

作成されたコードを読み出して引数を検索することをコンパイラがユーザに要求することはありません。コンパイラには，引数の正しい位置を指し示す$ARGnシンボルがあります。$ARG0 は， VAX システムの@AP+0 と同じもの，つまり引数の個数です。たとえば$ARG1 は最初の引数，$ARG2 は 2 番目の引数です。これらの引数は， CALL_ENTRY 指示文および JSB_ENTRY 指示文で定義されますが，EXCEPTION_ENTRY 指示文では定義されません。

C.11.4.3 $ARGn**シンボルを使用しない引数検索**

ユーザのコードで，コンパイラが$ARGnシンボルを作成しない追加の引数が使用されることがあります。.CALL_ENTRY ルーチンに対して定義される$ARGnシンボルの個数は，コンパイラが自動検出または MAX_ARGS によって検出した最大値または 16 のどちらか小さい方です。.JSB_ENTRY ルーチンの場合，引数は呼び出し元のスタック・フレーム内のホーム・ポジションにあり，コンパイラは実際の数を検出できないので，$ARGnシンボルを常に 8 個作成します。

ほとんどの場合，追加の引数は容易に見つけられますが，そうでない場合もあります。

C.11.4.4 **検索しやすい引数**

次の場合は，追加の引数は容易に見つけられます。

- 引数リストがホーム・ポジションにセットされていないで，$ARGnシンボルが$ARG7 またはそれより大きいと定義されている場合。引数リストがホーム・ポジションにセットされていない場合，$ARG7 以上の$ARGnシンボルは，スタック上でクォドワードとして引き渡されるパラメータのリストを常に指している。それ以降の引数は，最後に定義された$ARGnシンボルに続くクォドワードにある。
- 引数リストがホーム・ポジションにセットされていて，コンパイラが自動検出または MAX_ARGS によって検出した最大値以下の引数をチェックしたい場合。引数がホーム・ポジションにセットされていれば，$ARGnシンボルは，ホーム・ポジションにセットされた引数リストを常に指し示している。それ以降の引数は，最後に定義された$ARGnシンボルに続くクォドワードにある。

たとえば，次のようにすれば JSB ルーチン内の 8 番目の引数より先にある引数を調べることができます。この場合，引数リストのホーム・ポジションへのセットは呼び出し元で行うことが必要です。

```
DBG> EX $ARG8  ; highest defined $ARGn
.
.
.
DBG> EX .+4  ; next arg is in next longword
.
.
.
DBG> EX .+4  ; and so on
```

この例では，引数リストをホーム・ポジションにセットするときに呼び出し元が少なくとも 10 個の引数を検出したと想定しています。

引数のホーム・ポジションをセットしなかったルーチンの最後の$ARGnシンボルより先の引数を見つけるには，上記の例で EX .+4 を EX .+8 で置き換えた後，例に示すとおりに実行します。

C.11.4.5 検出しにくい引数

次の場合は，追加の引数を見つけるのは容易ではありません。

- 引数リストがホーム・ポジションにセットされていて，コンパイラが検出した数を超える引数を調べたい場合。$ARGnシンボルは，ホーム・ポジションにセットされた引数リストに格納されているロングワードを指している。コンパイラが転送する引数の数は，このリストで検出できる個数だけである。ホーム・ポジションにセットされた引数のうちの最後の引数の先のロングワードを調べると，各種の他のスタック・コンテキストを調べることになる。
- 引数リストがホーム・ポジションにセットされていないで，$ARGnシンボルが単に$ARG6 と定義されている場合。この場合，既存の$ARGnはレジスタを指し示すか，スタック・フレーム内のクォドワードの記憶位置を指し示すことになる。どちらの場合も，それ以降の引数は，定義された$ARGnシンボルの先のクォドワードの記憶位置を見ても調べることはできない。

これらのケースで追加の引数を見つける方法は，コンパイルされたマシン・コードを調べ，引数がどこに常駐しているかを判断することです。チェックしたい引数の最大値に対して MAX_ARGS が正しく指定されていれば，どちらの問題も解消します。

C.11.4.6 浮動小数点数データ付きのコードのデバッグ

Alpha システムで浮動小数点数データ付きの，コンパイルされた MACRO–32 コードをデバッグする際の重要な情報を次に説明します。

- Alpha 整数レジスタが浮動小数点数かどうか調べるには， EXAMINE/FLOAT コマンドが使用できる。

  Alpha システムに浮動小数点演算用レジスタが 1 セットあるとしても，浮動小数点演算を含む，コンパイルされた MACRO–32 コードでこれらのレジスタを使用することはない。使用されるのは Alpha 整数レジスタだけである。

コンパイルされた MACRO–32 コードでの浮動小数点数演算は，コンパイラの外部で動作するエミュレーション・ルーチンで実行される。したがって，たとえば R7 に MACRO–32 浮動小数点数演算を実行しても， Alpha の浮動小数点数レジスタ 7 には影響はない。

- .FLOAT 指示文または他の浮動小数点数記憶域指示文で宣言された記憶位置を調べるために EXAMINE コマンドを使用する際，デバッガは自動的にその値を浮動小数点数データとして表示する。
- G_FLOAT データ型を調べるために EXAMINE コマンドを使用する際，デバッガは自動的にその値を浮動小数点数データとして表示する。
- DEPOSIT コマンドを使用すれば， Alpha 整数レジスタに浮動小数点数データを格納できる。
- H_FLOAT はサポートされていない。

C.11.4.7 パック 10 進数データ付きのコードのデバッグ

Alpha システムでパック 10 進数データ付きの，コンパイルされた MACRO–32 コードをデバッグする際の重要な情報を次に説明します。

- .PACKED 指示文で宣言された記憶位置を調べるために EXAMINE コマンドを使用するときは，デバッガは自動的にその値をパック 10 進数データ型として表示する。
- パック 10 進数データを格納できる。構文は VAX の場合と同じである。

C.12 MACRO–64 (Alpha のみ)

以下の各節では，デバッガによる MACRO–64 のサポートについて説明します。

C.12.1 言語式の演算子

MACRO–64 言語には，高級言語と同じ意味での式というものはありません。受け入れられるのはアセンブリ時の式と，限られたセットの演算子だけです。デバッグ時に MACRO–64 のプログラミングで，他の言語の場合と同じくらい自由に式を使用できるようにするため，デバッガは， MACRO–64 自体には含まれていない多数の演算子を MACRO–64 の言語式の中で受け入れるようになっています。特に， BLISS 以後にモデル化された比較演算子とブール演算子のセットは完全に受け入れます。間接参照演算子と通常の算術演算子も受け入れられます。

種類	シンボル	機能
接頭辞	@	間接参照
接頭辞	.	間接参照
接頭辞	+	単項正符号

種類	シンボル	機能
接頭辞	–	単項負符号 (否定)
挿入辞	+	加算
挿入辞	–	減算
挿入辞	*	乗算
挿入辞	/	除算
挿入辞	MOD	剰余
挿入辞	@	左シフト
挿入辞	EQL	等値
挿入辞	EQLU	等値
挿入辞	NEQ	不等
挿入辞	NEQU	不等
挿入辞	GTR	大なり
挿入辞	GTRU	大なり符号なし
挿入辞	GEQ	以上
挿入辞	GEQU	以上符号なし
挿入辞	LSS	小なり
挿入辞	LSSU	小なり符号なし
挿入辞	LEQ	以下
挿入辞	LEQU	以下符号なし
接頭辞	NOT	ビット単位の NOT
挿入辞	AND	ビット単位の AND
挿入辞	OR	ビット単位の OR
挿入辞	XOR	ビット単位の排他的論理和
挿入辞	EQV	ビット単位の同値

C.12.2 言語式とアドレス式の構造

サポートされている， MACRO–64 の言語式とアドレス式の構造を次に示します。

シンボル	構造
<p,s,e>	BLISS 同様のビットフィールド

C.12.3 データ型

MACRO–64 は，データ型をラベル名にバインドします。バインドは，ラベル定義に続くアセンブラ指示文に従って行われます。たとえば，次のコード内の.LONG データ指示文は，ロングワード整数データ型をラベル V1， V2，および V3 にバインドするよう MACRO–64 に対して指示します。

```
.PSECT A, NOEXE
.BYTE 5
V1:
V2:
V3: .LONG 7
```

V1, V2, および V3 にバインドされている型を確認するには, SHOW SYMBOL /TYPE コマンドを V*パラメータ付きで実行します。実行結果の表示は次のとおりです。

```
data  .MAIN.\V1
   atomic type, longword integer, size: 4 bytes
data .MAIN.\V2
   atomic type, longword integer, size: 4 bytes
data .MAIN.\V3
   atomic type, longword integer, size: 4 bytes)
```

サポートされている MACRO–64 の指示文を次に示します。

MACRO–64 のデータ型	VMS のデータ型名
.BYTE	バイト符号なし (BU)
.WORD	ワード符号なし (WU)
.LONG	ロングワード符号なし (LU)
.SIGNED_BYTE	バイト整数(B)
.SIGNED_WORD	ワード整数(W)
.LONG	ロングワード整数(L)
.QUAD	クォドワード整数(Q)
.F_FLOATING	F 浮動小数点数(F)
.D_FLOATING	D 浮動小数点数(D)
.G_FLOATING	G 浮動小数点数(G)
.S_FLOATING (Alpha 固有)	S 浮動小数点数(S)
.T_FLOATING (Alpha 固有)	T 浮動小数点数(T)
(該当なし)	パック 10 進数(P)

C.13 PASCAL

以下の各サブトピックでは，デバッガによる Pascal のサポートについて説明します。

C.13.1 言語式の演算子

言語式でサポートされている Pascal の演算子を次に示します。

種類	シンボル	機能
接頭辞	+	単項正符号
接頭辞	–	単項負符号 (否定)
挿入辞	+	加算，連結
挿入辞	–	減算
挿入辞	*	乗算
挿入辞	/	実数の除算
挿入辞	DIV	整数の除算
挿入辞	MOD	モジュロ
挿入辞	REM	剰余
挿入辞	IN	集合メンバシップ
挿入辞	=	等値
挿入辞	<>	不等
挿入辞	>	大なり
挿入辞	>=	以上
挿入辞	<	小なり
挿入辞	<=	以下
接頭辞	NOT	論理否定
挿入辞	AND	論理積
挿入辞	OR	論理和

型キャスト演算子(::)は，言語式ではサポートされていません。

C.13.2 言語式とアドレス式の構造

サポートされている， Pascal の言語式とアドレス式の構造を次に示します。

シンボル	構造
[ ]	添字指定
.(ピリオド)	レコードの構成要素の選択
^(サーカンフレックス)	ポインタの間接参照

C.13.3 定義済みのシンボル

サポートされている， Pascal の定義済みのシンボルを次に示します。

シンボル	意味
TRUE	論理値 TRUE
FALSE	論理値 FALSE
NIL	NIL ポインタ

C.13.4 組み込み関数

サポートされている， Pascal の組み込み関数を次に示します。

シンボル	意味
SUCC	論理的後続データ
PRED	論理的先行データ

C.13.5 データ型

サポートされている Pascal のデータ型を次に示します。

Pascal のデータ型	VMS のデータ型名
INTEGER	ロングワード整数(L)
INTEGER	ワード整数 (W,WU)
INTEGER	バイト整数 (B,BU)
UNSIGNED	ロングワード符号なし (LU)
UNSIGNED	ワード符号なし (WU)
UNSIGNED	バイト符号なし (BU)
SINGLE， REAL	F 浮動小数点数(F)
REAL(Alpha および Integrity 固有)	IEEE S 浮動小数点数(FS)
DOUBLE	D 浮動小数点数(D)
DOUBLE	G 浮動小数点数(G)
DOUBLE(Alpha および Integrity 固有)	IEEE T 浮動小数点数(FT)
QUADRUPLE(VAX および Integrity 固有)	H 浮動小数点数(H)
BOOLEAN	(なし)
CHAR	ASCII テキスト(T)
VARYING OF CHAR	変形テキスト (VT)
SET	(なし)
FILE	(なし)
列挙	(なし)
部分範囲	(なし)
型付きポインタ	(なし)
配列	(なし)
レコード	(なし)
可変レコード	(なし)

Pascal の言語式の中では，デバッガは，[1,2,5,8..10]や[RED,BLUE]といった Pascal の集合定数を受け入れます。

REAL 型の浮動小数点数は，コンパイラのスイッチまたはソース・コードの属性によって， F 浮動小数点数または IEEE S 浮動小数点数のどちらかで表現されます。

DOUBLE 型の浮動小数点数は，コンパイラのスイッチまたはソース・コードの属性によって， D 浮動小数点数， G 浮動小数点数，または IEEE T 浮動小数点数のどれかで表現されます。

C.13.6 補足情報

通常は，変数，レコード・フィールド，配列の構成要素のそれぞれに対して，検査，評価，格納を行うことができますが，次の場合は例外です。プログラムの中で変数が参照されていない場合， Pascal コンパイラは変数を割り当てません。変数を割り当てていないときにその変数を検査しようとしたり，その変数へ格納しようとするとエラー・メッセージが表示されます。

変数にデータを格納する場合，格納される値が変数よりも大きいときは，デバッガは上位ビットを切り捨てます。格納される値が変数よりも小さいときは，上位ビットをゼロで埋めます。代入互換性の規則に違反した格納をすると情報メッセージが表示されます。

実行中のブロック内であればどのブロックでも自動変数を調べたり，自動変数へ格納することができます。しかし，自動変数はスタック領域に割り当てられ，レジスタに格納されるので，変数が初期化されるまで，または変数に値が代入されるまで，自動変数の値は未定義であるとみなされます。

C.13.7 制限事項

Pascal についてのデバッガの制限事項は次のとおりです。

VARYING OF CHAR の文字列の値を調べることはできますが， Pascal の通常の構文を使用して.LENGTH フィールドや.BODY フィールドの値を調べることはできません。たとえば， VARS が文字列変数名の場合，次のコマンドはサポートされません。

```
DBG> EXAMINE VARS.LENGTH
DBG> EXAMINE VARS.BODY
```

これらのフィールドを確認するには，次の方法を使用します。

使用する	使用しない
EXAMINE/WORD VARS	EXAMINE VARS.LENGTH
EXAMINE/ASCII VARS+2	EXAMINE VARS.BODY

C.14 PL/I (Alpha **のみ**)

次の各サブトピックでは，デバッガによる PL/I のサポートについて説明します。

C.14.1 言語式の演算子

言語式でサポートされている PL/I の演算子を次に示します。

種類	シンボル	機能
接頭辞	+	単項正符号
接頭辞	–	単項負符号 (否定)
挿入辞	+	加算
挿入辞	–	減算
挿入辞	*	乗算
挿入辞	/	除算
挿入辞	**	べき乗
挿入辞	｜｜	連結
挿入辞	=	等値
挿入辞	^=	不等
挿入辞	>	大なり
挿入辞	>=	以上
挿入辞	^<	以上
挿入辞	<	小なり
挿入辞	<=	以下
挿入辞	^>	以下
接頭辞	^	ビット単位の NOT
挿入辞	&	ビット単位の AND
挿入辞	｜	ビット単位の OR

C.14.2 言語式とアドレス式の構造

サポートされている， PL/I の言語式とアドレス式の構造を次に示します。

シンボル	構造
( )	添字指定
.(ピリオド)	構造体の構成要素の選択
->	ポインタの間接参照

C.14.3 データ型

サポートされている PL/I のデータ型を次に示します。

PL/I のデータ型	VMS のデータ型名
FIXED BINARY	バイト整数(B)，ワード整数 (W)，またはロングワード整数(L)
FIXED DECIMAL	パック 10 進数(P)
FLOAT BIN/DEC	F 浮動小数点数(F)
FLOAT BIN/DEC	D 浮動小数点数(D)
FLOAT BIN/DEC	G 浮動小数点数(G)
BIT	ビット(V)
BIT	ビット境界合わせなし (VU)
CHARACTER	ASCII テキスト(T)
CHARACTER VARYING	変形テキスト (VT)
FILE	(なし)
ラベル	(なし)
ポインタ	(なし)
配列	(なし)
構造体	(なし)

C.14.4 静的変数と非静的変数

次の記憶クラスの変数は静的に割り当てられます。

STATIC
EXTERNAL
GLOBALDEF
GLOBALREF

次の記憶クラスの変数は，スタック上またはレジスタ内に非静的に割り当てられます。

AUTOMATIC
BASED
CONTROLLED
DEFINED
PARAMETER

C.14.5 データの検査と操作

以下の各サブトピックでは， PL/I のデータ型を使用した EXAMINE コマンドの例を示します。デバッガのサポートで PL/I 固有の事項についても説明しています。

C.14.5.1 EXAMINE コマンドの例

次に， EXAMINE コマンドでのいくつかの PL/I のデータ型の使用例を示します。

- FIXED DECIMAL (10,5) として宣言されている変数の値を調べます。

```
DBG> EXAMINE X
PROG4\X:    540.02700
```

- 構造体変数の値を表示します。

```
DBG> EXAMINE PART
MAIN_PROG\INVENTORY_PROG\PART
    ITEM:       "WF-1247"
    PRICE:        49.95
    IN_STOCK:     24
```

- ピクチャ変数の値を調べます。デバッガは値を二重引用符で囲んで表示するので注意してください。

```
DBG> EXAMINE Q
MAIN\Q:    "666.3330"
```

- ポインタの値 (ポインタがアクセスする変数の仮想アドレス) を調べ，その値を省略時設定の 10 進数ではなく， 16 進数で表示します。

```
DBG> EXAMINE/HEXADECIMAL P
PROG4\SAMPLE.P:  0000B2A4
```

- BASED 属性を持った変数の値を調べます。ここでは，変数 X のポインタである PTR を使用して，変数 X を BASED(PTR) として宣言しています。

```
DBG> EXAMINE X
PROG\X:    "A"
```

- BASED として宣言されている変数 X の値を， POINTER として宣言されている変数 PTR を使用して調べます。ここでは前の例のように X が BASED(PTR) として宣言されているのではなく， PL/I コードの後続行で PTR を X と対応づけています。

```
ALLOCATE X SET (PTR);
```

この場合， X の値は次のように調べられます。

```
DBG> EXAMINE PTR->X
PROG6\PTR->X:    "A"
```

C.14.5.2 デバッガのサポートについての注意事項

デバッガによる PL/I のサポートについては次の点に注意してください。

入口変数やラベル変数で，または入口やラベルの形式で DEPOSIT コマンドを使用することはできません。また，配列全体や構造体全体といっしょに DEPOSIT コマンドを使用することもできません。入口変数やラベル変数で，または入口やラベルの形式

で EXAMINE コマンドを使用することはできません。 EXAMINE コマンドの代わりに EVALUATE/ADDRESS コマンドを使用してください。

GLOBALDEF VALUE リテラルなどのグローバル・リテラルの属性や値を確認するために EXAMINE コマンドを使用することはできません。これはグローバル・リテラルが静的な式であるためです。代わりに EVALUATE コマンドを使用してください。

コンパイル時の変数およびプロシージャといっしょに， EXAMINE， EVALUATE， DEPOSIT の各コマンドを使用することはできません。しかし，その定数がデスティネーションではなくソースである場合は，コンパイル時の定数とともに EVALUATE コマンドや DEPOSIT コマンドを使用することができます。 EXAMINE コマンドは使用できません。

初期化されていない自動変数の内容は，値が代入されるまでは無効です。代入する前に検査した場合，どのような値が表示されるかは予測できません。

他のポインタの値をポインタ変数に格納することで両方のポインタへのシンボル参照を行うか，または仮想アドレスをポインタ変数に格納するかのどちらかの方法によって，ポインタ変数へ値を格納することができます。 EVALUATE/ADDRESS コマンドを使用すると，変数の仮想アドレスを知ることができます。それからそのアドレスをポインタに格納してください。ポインタをチェックする場合，デバッガは，ポインタが指す変数の仮想アドレスの形式でポインタの値を表示します。

PL/I の言語規則に準拠するために， PL/I の言語式の*n*または*n.n*の形式の数値定数はすべて整数定数や浮動小数点定数ではなく，パック 10 進数の定数として処理されます。したがって， 10 の内部表現は 0C01h であり， 0Ah ではありません。

nEn または n.nEn の構文を使用すると，浮動小数点定数を入力することができます。

内部表現がロングワード整数である定数を入力する構文は， PL/I にはありません。デバッガは PL/I の型変換規則をサポートしているので，通常，デバッグ時にこの制約が重大なものになることはありませんが，整数定数を入力することはできます。入力にはデバッガの%HEX，%OCT，%BIN の各演算子を使用してください。これが可能なのは， 10 進数以外の定数は FIXED BINARY とみなされるためです。たとえば， EVALUATE/HEXADECIMAL 53 + %HEX 0というコマンドは 00000035 を表示します。

C.15 UNKNOWN 言語

次の各サブトピックでは，UNKNOWN である言語の，デバッガのサポートについて説明します。

C.15.1 言語式の演算子

UNKNOWN の言語について，言語式でサポートされている演算子を次に示します。

種類	シンボル	機能
接頭辞	+	単項正符号
接頭辞	–	単項負符号 (否定)
挿入辞	+	加算
挿入辞	–	減算
挿入辞	*	乗算
挿入辞	/	除算
挿入辞	&	連結
挿入辞	//	連結
挿入辞	=	等値
挿入辞	<>	不等
挿入辞	/=	不等
挿入辞	>	大なり
挿入辞	>=	以上
挿入辞	<	小なり
挿入辞	<=	以下
挿入辞	EQL	等値
挿入辞	NEQ	不等
挿入辞	GTR	大なり
挿入辞	GEQ	以上
挿入辞	LSS	小なり
挿入辞	LEQ	以下
接頭辞	NOT	論理否定
挿入辞	AND	論理積
挿入辞	OR	論理和
挿入辞	XOR	排他的論理和
挿入辞	EQV	同値

C.15.2 言語式とアドレス式の構造

UNKNOWN の言語について，サポートされている言語式とアドレス式の構造を次に示します。

シンボル	構造
[ ]	添字指定
( )	添字指定
.(ピリオド)	レコードの構成要素の選択
^(サーカンフレックス)	ポインタの間接参照

C.15.3 定義済みのシンボル

UNKNOWN の言語について，サポートされている定義済みのシンボルを次に示します。

シンボル	意味
TRUE	論理値 TRUE
FALSE	論理値 FALSE
NIL	NIL ポインタ

C.15.4 データ型

言語が UNKNOWN に設定されている場合，デバッガは，他の言語で受け入れられるデータ型をすべて認識しますが，ピクチャ型やファイル型など，限られたいくつかの言語固有のデータ型は認識しません。 UNKNOWN の言語式では， OpenVMS 呼び出し規則のスカラ・データ型のほとんどを使用することができます。

- UNKNOWN の言語では，レコードの構成要素の選択にドット表記法を使用することができます。たとえば， C がレコード B の構成要素であり，その B がレコード A の構成要素である場合， A.B.C という表記で C を参照することができます。また，配列のどの構成要素にも添字を付けることができます。たとえば B が配列であるとすると， A.B[2,3].C で C を参照することができます。
- UNKNOWN の言語では，丸括弧と角括弧の両方を添字の括弧に使用することができます。たとえば， A[2,3]と A(2,3) は等価です。

D

EIGHTQUEENS.C

この付録では，第 8 章，第 9 章，および第 10 章の各章にある多くの図で使用されるプログラムのソース・コード， EIGHTQUEENS.C および 8QUEENS.C を示します。ここで示すプログラムは，上記の各章で説明されているプロシージャを理解するためだけのものです。

D.1 EIGHTQUEENS.C

Example D–1 に， EIGHTQUEENS.C を示します。これは， eightqueens の問題を解決するための，シングル・モジュール構成のプログラムです。

Example D–1 **シングル・モジュール構成のプログラム** EIGHTQUEENS.C

```
extern void setqueen();
extern void removequeen();
extern void trycol();
extern void print();
    int a[8];     /* a : array[1..8]  of boolean */
    int b[16];    /* b : array[2..16] of boolean */
    int c[15];    /* c : array[-7..7] of boolean */
    int x[8];

/* Solve eight-queens problem */
main()
{
    int i;
    for (i=0; i <=7; i++)
 a[i] = 1;
    for (i=0; i <=15; i++)
 b[i] = 1;
    for (i=0; i <=14; i++)
 c[i] = 1;
    trycol( 0 );
} /* End main */
```

(次ページに続く)

Example D–1 (続き) シングル・モジュール構成のプログラム EIGHTQUEENS.C

```
void trycol( j )
    int j;
{
    int m;
    int safe;
    m = -1;
    while (m++ < 7)
       {
       safe = (a[m] ==1) && (b[m + j] == 1) && (c[m - j + 7] ==1);
       if (safe)
            {
     setqueen(m, j);
     x[j] = m + 1;
     if (j < 7)
  trycol(j + 1);
     else
  print();
     removequeen(m, j);
     }
       }
} /* End trycol */

void setqueen(m, j)
    int m;
    int j;
{
    a[m] = 0;
    b[m + j] = 0;
    c[m - j + 7] = 0;
} /* End setqueen */

void removequeen(m, j)
    int m;
    int j;
{
    a[m] = 1;
    b[m + j] = 1;
    c[m - j + 7] = 1;
} /* End removequeen */

void print()
{
    int k;
    for (k=0; k<=7; k++)
 {
 printf(" %d", x[k]);
 }
    printf("\n");
} /* End print */
```

D.2 8QUEENS.C

8QUEENS.C は， eightqueens の問題を解決するためのマルチ・モジュール構成のプログラムです。このプログラムは， 8QUEENS.C (Example D–2) および 8QUEENS_SUB.C (Example D–3) の 2 つのモジュールから構成されます。

Example D–2 8QUEENS.C **のメイン・モジュール**

```
extern void trycol();
    int a[8];     /* a : array[1..8]  of boolean */
    int b[16];    /* b : array[2..16] of boolean */
    int c[15];    /* c : array[-7..7] of boolean */
    int x[8];

main()    /* Solve eight-queens problem */
{
    int i;
    for (i=0; i <=7; i++)
 a[i] = 1;
    for (i=0; i <=15; i++)
 b[i] = 1;
    for (i=0; i <=14; i++)
 c[i] = 1;
    trycol(0);
    printf(" Solved eight-queens problem!\n");
} /* End main */
```

Example D–3 8QUEENS_SUB.C **のサブ・モジュール**

```
extern int a[8];
extern int b[16];
extern int c[15];
extern void setqueen();
extern void removequeen();
extern void print();

    int x[8];
```

(**次ページに続く**)

Example D–3（続き） 8QUEENS_SUB.C のサブ・モジュール

```
void trycol( j )
    int j;
{
    int m;
    int safe;
    m = -1;
    while (m++ < 7)
       {
       safe = (a[m] ==1) && (b[m + j] == 1) && (c[m - j + 7] ==1);
       if (safe)
            {
            setqueen(m, j);
            x[j] = m + 1;
            if (j < 7)
                trycol(j + 1);
            else
                print();
            removequeen(m, j);
            }
       }
}       /* End trycol */

void setqueen(m, j)
    int m;
    int j;
{
    a[m] = 0;
    b[m + j] = 0;
    c[m - j + 7] = 0;
}       /* End setqueen */

void removequeen(m, j)
    int m;
    int j;
{
    a[m] = 1;
    b[m + j] = 1;
    c[m - j + 7] = 1;
}       /* End removequeen */

void print()
{
    int k;
    for (k=0; k<=7; k++)
        {
        printf(" %d", x[k]);
        }
    printf("\n");
}       /* End print */
```

索引

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

エ

オ

カ

キ

ク

ケ

コ

サ

シ

ス

セ

ソ

タ

テ

ト

ナ

ニ

ネ

ハ

ヒ

フ

ヘ

ホ

マ

メ

モ

ユ

ヨ

ラ

リ

ル

レ

ロ

ワ

OpenVMS **デバッガ説明書**

2010 年 10 月　発行

日本ヒューレット・パッカード株式会社

〒 140-8641 東京都品川区東品川 2-2-24 天王洲セントラルタワー

電話　(03)5463-6600 (大代表)

AA–PY9GF–TE.2