Titel:
Title:

UCL Debugger User Manual

Dokumenten Typ:
Document Type: User Manual

Dokumentenklasse:
Document Class:

Klassifikations-Nr.:
Classification No.:

Dokumentenkategorie:
Document Category:

Konfigurations-Nr.:
Configuration Item No.:

Produktklassifizierungs-Nr.:
Classifying Product Code:

Freigabe Nr.:
Release No.:

Bearbeitet: Prepared by:	Franz Kruse	**Org. Einh.:** Organ. Unit:	TE 55	**Unternehmen:** Company:	EADS ST Bremen
Geprüft: Agreed by:	Stephan Marz	**Org. Einh.:** Organ. Unit:	TE 55	**Unternehmen:** Company:	EADS ST Bremen
Genehmigt: Approved by:	Jürgen Frank	**Org. Einh.:** Organ. Unit:	TE 55	**Unternehmen:** Company:	EADS ST Bremen
Genehmigt: Approved by:		**Org. Einh.:** Organ. Unit:		**Unternehmen:** Company: Agency:	

Copyright by EADS - All Rights Reserved

Attribut-Liste/List of Attributes

Vertrags Nr.:
Contract No.:

Dokument Ref.Nr.:
Document Ref.No.:

Lieferbedingungs Nr.:
DRL/DRD No.:

Seitenzahl Dokument-Hauptteil:
Pages of Document Body: ?

Schlagwörter:
Headings:

UCL
Debugger

Erstellungssystem:
S/W Tool:

Kurzbeschreibung:
Abstract:

This document describes the usage of the UCL Debugger.

Copyright by EADS - All Rights Reserved

DCR Daten/Dokument-Änderungsnachweis/Data/Document Change Record

Überarbeitung Revision	**Datum** Date	**Betroffener Abschnitt/Paragraph/Seite** Affected Section/Paragraph/Page	**Änderungsgrund/Kurze Änderungsbeschreibung** Reason for Change/Brief Description of Change

Copyright by EADS - All Rights Reserved

TOC-Inhaltsverzeichnis/Table of Contents

Copyright by EADS - All Rights Reserved

1. Introduction

1.1 Identification

This is the UCL Debugger User Manual, document number CGS-RIBRE-MA-0001.

1.2 Purpose

This document describes the usage of the UCL Debugger. It assumes the user is familiar with the User Control Language (UCL) described in [2.2.2]. For usage on purely graphical level, using windows, buttons and the mouse, knowledge of UCL is sufficient.

But since the UCL Debugger provides a command subwindow for direct commanding in HLCL, a knowledge of the High Level Command Language (HLCL) is desirable as well. If automated command sequences are to be used, ia good knowledge of HLCL is vital. For a description of HLCL see [2.2.3].

For an expert user it may sometimes be useful to understand details of execution on I-Code level within the Virtual Stack Machine. A description of the stack machine architecture and I-Code is given in [2.2.4].

Copyright by EADS - All Rights Reserved

2. Applicable and Reference Documents

2.1 Applicable Documents

none

2.2 Reference Documents

2.2.1 CGS User Manual
CGS-RIBRE-SUM-0001, Issue 3/-, 2004-04-29

2.2.2 User Control Language (UCL) Reference Manual
CGS-RIBRE-STD-0001, Issue 2/A, 2004-09-01

2.2.3 High Level Command Language (HLCL) Reference Manual
CGS-RIBRE-STD-0002, Issue 2/A, 2004-09-01

2.2.4 UCL Virtual Stack Machine and I-Code Reference Manual
CGS-RIBRE-STD-0003, Issue 2/A, 2004-09-01

Copyright by EADS - All Rights Reserved

3. Overview

The UCL Debugger allows to debug Automated Procedures, programs written in UCL on source code level within a graphical window environment. It provides the usual debugging features like single-step execution, breakpoints, reading and writing variables etc. Besides the graphical environment, it provides an HLCL command subwindow that makes nearly all debugging funtionality available on command level and, furthermore, allows to use the normal HLCL features. In particular, the user may automat parts or all of a debugging session through HLCL command sequences.

The UCL Debugger is a distributed application: The debugger kernel comprising the user interface and all high level debugging functionality, resides in a workstation, while the AP being debugged is executed by an I-Code interpreter in the CGS execution environment normally located in a different computer. The I-Code interpreter executes low level debugging functions on request of the debugger kernel. Since this distributed execution requires extensive network communication, the debugger will execute an AP considerably slower than when executed directly.

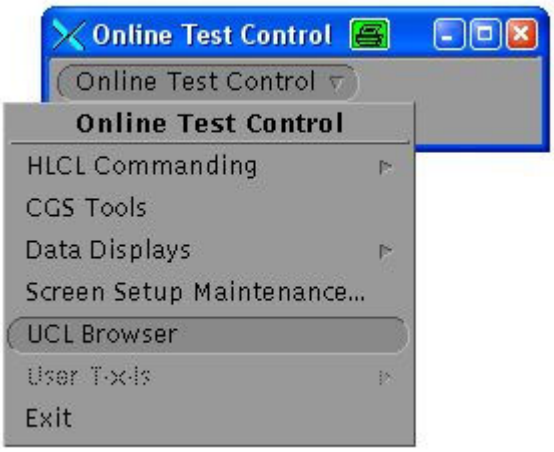
The following chapters describe both the graphical user interace and the HLCL command interface, as well as specific debugger functionality that requires special care by the user.

Copyright by EADS - All Rights Reserved

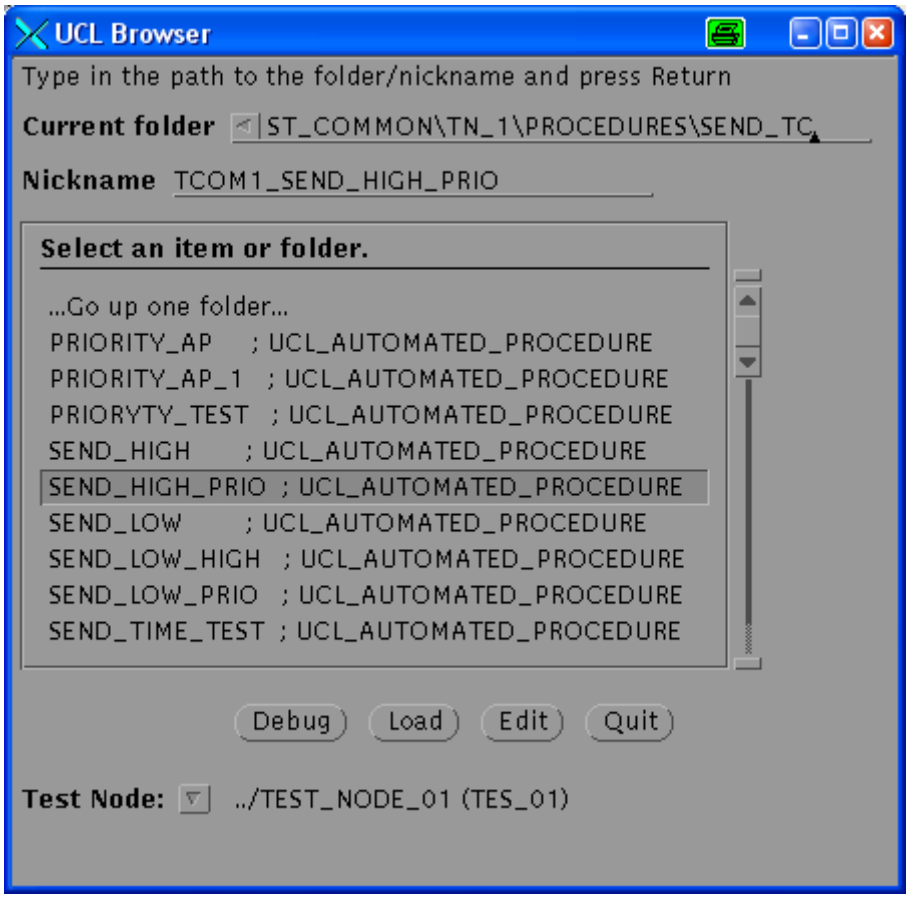
4. Starting the UCL Debugger

4.1 Starting the Debugger from within HCI

The UCL Debugger is usually started from within the UCL Browser in the CGS Online Test Control environment (HCI), for a description see the CGS User Manual [2.2.1].



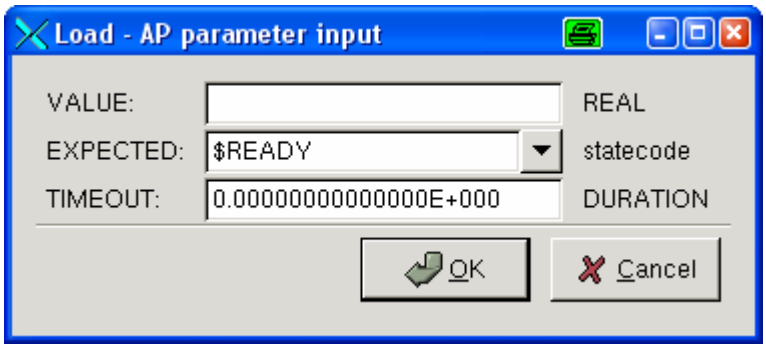
Open the UCL Browser
from the HCI selector window.



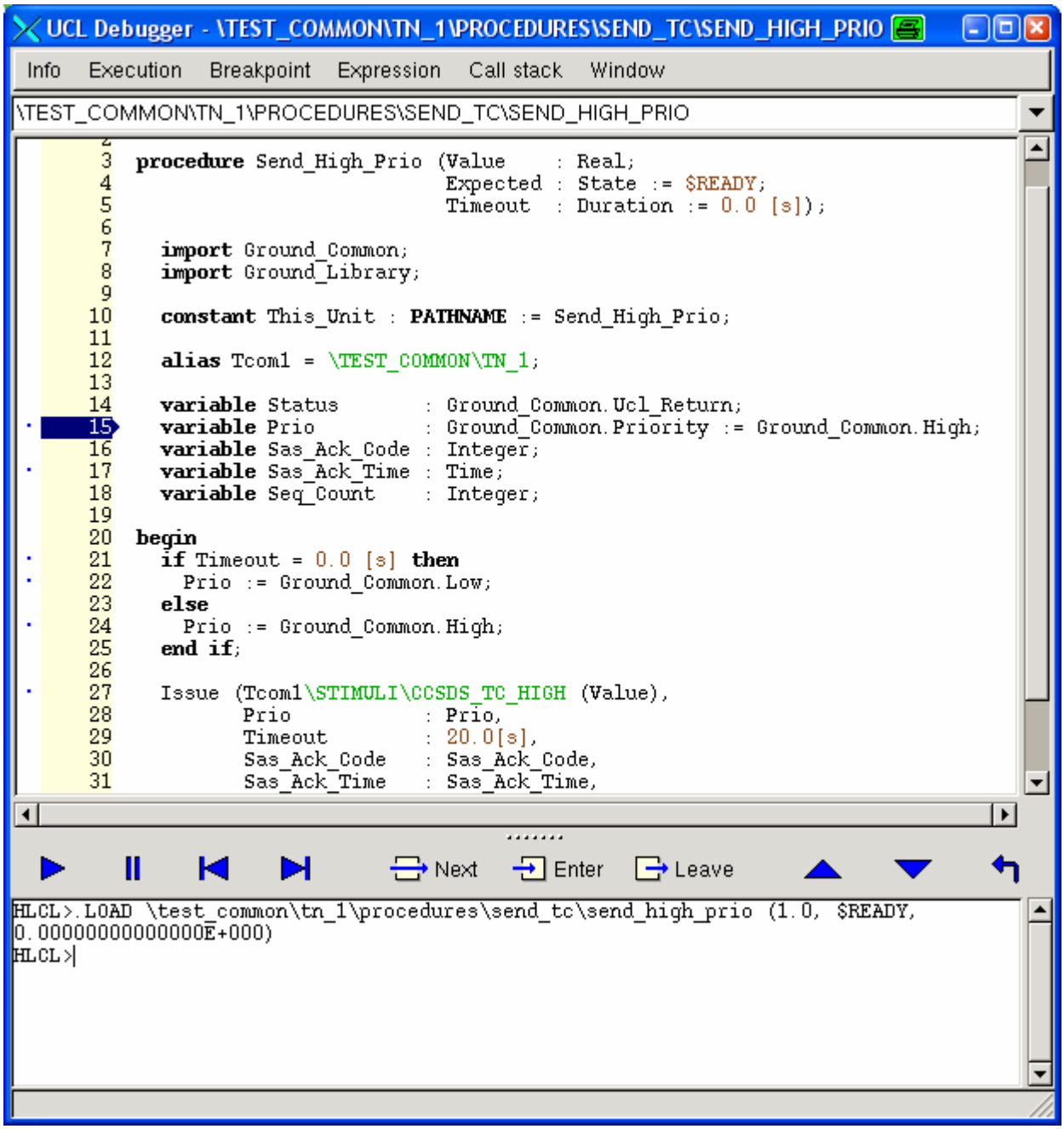
Navigate to the AP to be debugged, select it and click the Debug button.

Copyright by EADS - All Rights Reserved

The UCL Debugger window will start, it will automatically connect to the appropriate test node and load the selected AP. If the AP has parameters, a pop-up window will appear that requests you to enter parameter values. Any default values are already inserted. For data types with a limited set of values, combo boxes allow you to select the desired value from the set of available values.



Insert the parameter values and click on OK, the UCL Debugger window appears with the AP loaded.



Copyright by EADS - All Rights Reserved

4.2 Starting the Debugger from the Command Line

The UCL Debugger may also be started from a UNIX command line in a shell window. The `ucl_debugger` program is located in either of the architecture specific directories

```
$CLS_HOME/bin/linuxi     for Linux
$CLS_HOME/bin/sun5       for SunOS
```

It has a simple command syntax:

```
ucl_debugger [parameters] [-options]
```

A short parameter/option overview can be obtained with the `-help` option:

```
ucl_debugger
     [<item>]                          <- item to be debugged (may include pa-
rameter list)
     [-help]                           <- print this help, then exit
     [-environment <value>...="$MDA_ENVIRONMENT"]
     [-node <value>="TES_01"]          <- test node identifier
     [-login_sequence <value>]         <- login command sequence
     [-sequence <value>]               <- command sequence to be executed in
batch mode
     [-font_name <value>="Lucidatypewriter"]  <- text font name
     [-font_size <value>="10"]         <- text font size
     [-logfile <value>]                <- log file name
```

Parameter

`item`

the item (AP) to be debugged. It can be given as either the pathname or the nickname in the qualified form `\.`*nickname*. If the AP has parameters, the actual parameter list can be given in parentheses after the name:

```
'name (parameter_1, ..., parameter_n)'
```

The parameter list is written in usual UCL syntax, in positional or named notation. Optional parameters may be omitted. The whole `item` parameter should be enclosed in quotes, in order to keep it together as one parameter for the shell and to mask the backslashes from the shell.
If the parameter list is omitted, parameter input will be requested with a pop-up window, like shown above. If the `item` parameter is omitted altogether, the debugger will start without an AP loaded, you will then have to load an AP within the debugger window.

Options

Option names can be written in lower, upper or mixed case, and they may be abbreviated by dropping characters from the end of the name, as long as the name remains unique. Name parts separated by underscores can be abbreviated separately (e. g `-log` or `-log_seq` are valid abbreviations for `-login_sequence`).

`-environment` *environment*

defines the database user environment. If omitted, the environment will be obtained from the environment variable `MDA_ENVIRONMENT` in the same syntactic form:

```
<environment> ::=
  CCU <element_config> <mission> <system_tree_version> <ccu> |
  CDU element_config> <mission> <system_tree_version> <ccu>

<ccu> ::=
  <pathname> <ccu_name> <version>.<issue>.<revision>

<cdu> ::=
  <pathname> <version>.<issue>.<revision> <test_version> <instance>
```

Example:

```
-environment "CCU APM MASTER 4 \APM\FLTSYS FLAP_TEST 1.0.0"
```


Copyright by EADS - All Rights Reserved

Options **(continued)**

`-node` *name*

the name of the test node where the I-Code interpreter runs. Default: `TES_01`

`-login_sequence` *sequence*

the name of an HLCL command sequence (pathname, qualified nickname or file name) that is to be run as a login sequence, i. e. before user interaction starts

`-sequence` *sequence*

the name of an HLCL command sequence (pathname, qualified nickname or file name) that is to be run in batch mode. When the sequence has terminated, the debugger remains loaded for interactive commanding.

`-font_name` *font*

the name of a text font to be used in the debugger source and command window. Default: `Lucidatypewriter`

`-font_size` *size*

the size of the text font in points. Default: `10`

`-log_file` *file*

the name of a file to which the debugger logs all user interaction and debugger responses. If omitted, no logging is performed.

Examples of program calls

```
ucl_debugger "\.TEST_TC" -env "CCU APM MASTER 4 \APM\FLTSYS FLAP_TEST 1.0.0"
```

Load the AP with nickname `TEST_TC` on the default test node. Give an explicit database environment.

```
ucl_debugger "\.TEST_TC" -node TES_02
```

Load the AP with nickname `TEST_TC` on the test node `TES_02`. Use the database environment defined in the environment variable `MDA_ENVIRONMENT`.

```
ucl_debugger "\.TEST_TC" -node TES_02 -log_file debugger.log
```

Same as previous command, in addition create a log file.

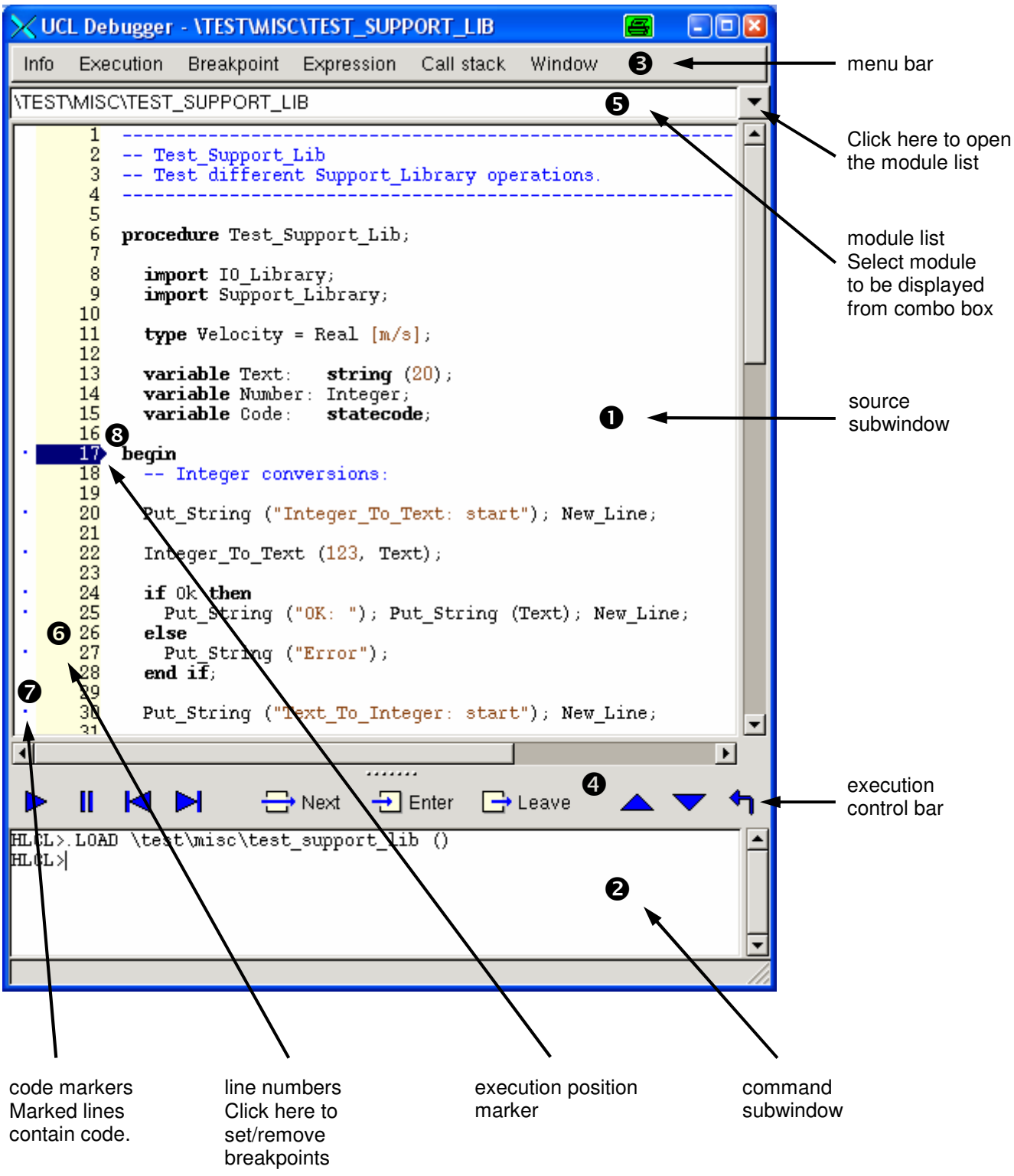
```
ucl_debugger
```

Just start the debugger on the default node and with the database environment in `MDA_ENVIRONMENT`. An AP will be loaded interactively within the debugger window.

Copyright by EADS - All Rights Reserved

5. The Debugger Main Window

5.1 Overview



Copyright by EADS - All Rights Reserved

The components of the debugger main window:

Some of these components are described in more detail in the following chapters.

❶ **source subwindow**

This subwindow displays the source text of the compilation unit currently being executed.

❷ **command subwindow**

This subwindow is an HLCL command window. Most of the debugger control functions done with the mouse are translated into HLCL commands and executed here, see 6.

❸ **menu bar**

The menu bar with different menus for various debugger functionality., see 5.2

❹ **execution control bar**

This is a tool bar with the most frequently used functions for execution control, such as start, stop, continue, single step etc. All these functions are also available through the menu bar, but for convenience they have been put in a separate and easy-to-access tool bar, as well, see 5.3.

❺ **module list**

All modules (the AP itself and all user libraries linked to the AP) are listed in this combo box. The name of the currently displayed module is shown in the text field. You can display the source text of other modules by just selecting a different module from the list.

❻ **line numbers**

The line numbers of the source lines are displayed in s separate row. Clicking on a line number sets or removes, resp., a breakpoint on that line.

❼ **code markers**

Lines that contain executable I-Code are marked with small blue dots left to the line number.

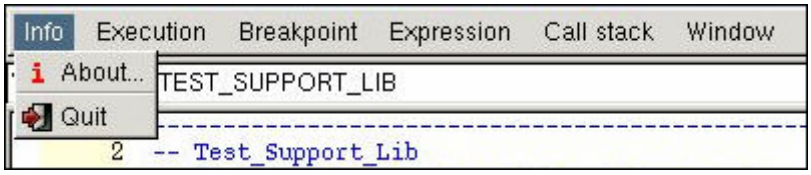
❽ **execution position marker**

The blue arrow on the line number points to the line that will be executed with the next single step.

Copyright by EADS - All Rights Reserved

5.2 The Menu Bar

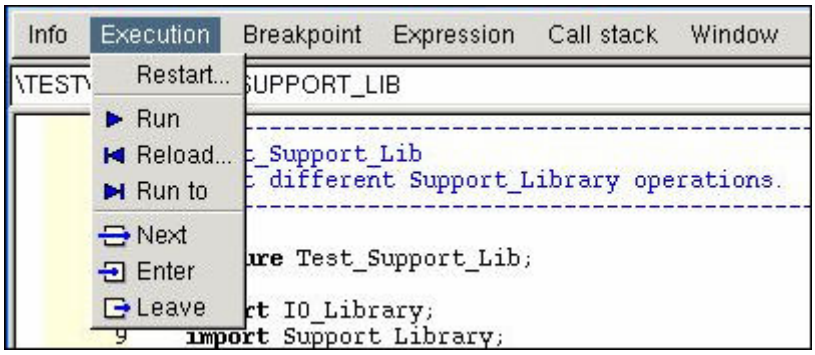
5.2.1 The *Info* Menu



About... Display short information about the program version.

Quit Quit the debugger.

5.2.2 The *Execution* Menu



Start.../ Restart... When no AP is loaded, the menu item *Start...* will be shown, when an AP is loaded, it is *Restart...* This menu item allows you to start or to restart an AP with an explicit parameter list.
Executed command: `.LOAD AP_name (parameters)`

Run Start or continue execution up to the next breakpoint. Alternatively, you can use the execution control bar, see 5.3.
Executed command: `.RUN`

Reload... Reload the current AP with its original parameter list. The AP is now in initial state, and execution may be started from the beginning. Alternatively, you can use the execution control bar, see 5.3.
Executed command: `.LOAD AP_name (parameters)`

Run to Continue execution up to the marked line. In order to mark the line, mark at least one character in the line. Alternatively, you can use the context menus, see 5.4 + 5.5.
Executed command: `.RUN line_number`

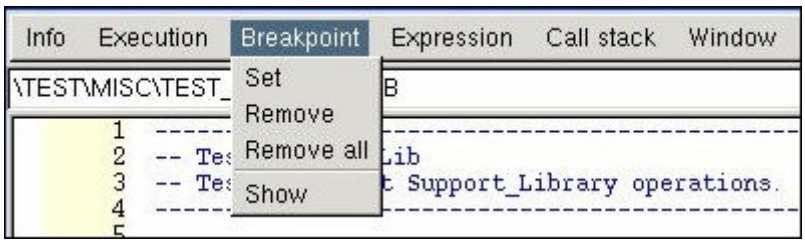
Next Execute one source line, step over subprograms, i. e. if the line contains a procedure or function call, do not stop in the subprogram, but execute it completely as part of the line. Alternatively, you can use the execution control bar, see 5.3
Executed command: `.NEXT`

Enter Execute one source line, step into subprograms, i. e. if the line contains a procedure or function call, stop on the first executable instruction within the subprogram. Alternatively, you can use the execution control bar, see 5.3
Executed command: `.ENTER`

Leave Leave the current subprogram, i. e. continue execution until the subprogram returns to the caller and stop after the call. Alternatively, you can use the execution control bar, see 5.3
Executed command: `.LEAVE`

Copyright by EADS - All Rights Reserved

5.2.3 The *Breakpoint* Menu



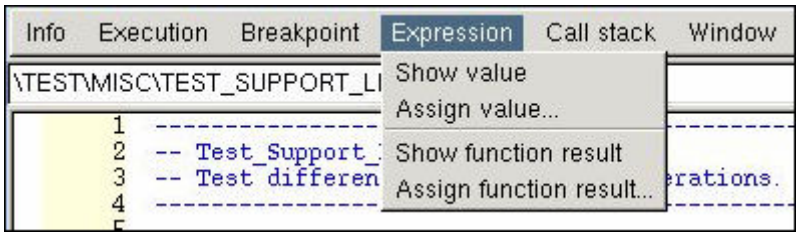
Set — Set a breakpoint on the marked line (at least one character marked in the line). Alternatively, you can use the context menus, see 5.4 + 5.5, or just click on the line number.
Executed command: `.BREAKPOINT line_number`

Remove — Remove the breakpoint from the marked line (at least one character marked in the line). Alternatively, you can use the line context menus, see 5.4 + 5.5, or just click on the line number.
Executed command: `.REMOVE line_number`

Remove all — Remove all breakpoints.
Executed command: `.REMOVE 0`

Show — Display a list of breakpoints currently set.
Executed command: `.BREAKPOINT`

5.2.4 The *Expression* Menu



Show value — Display the value of the marked expression. The expression may be a single item or a complex expression, including software variables, but it must not contain function calls. Alternatively, you can use the source context menu, see 5.4.
Executed command: `?? (expression)`

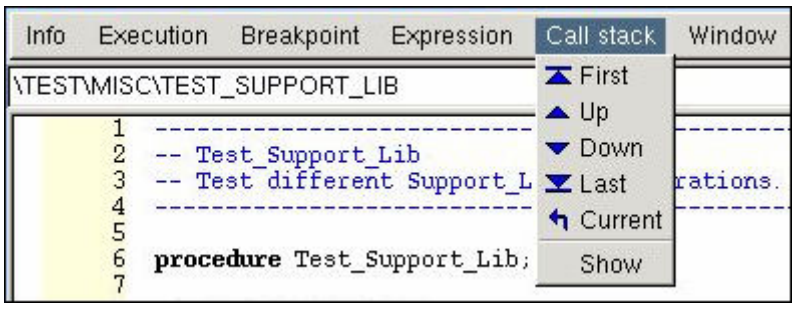
Assign value... — Assign a value to the marked variable or software variable. A pop-up dialog will prompt you for the value to be assigned. Alternatively, you can use the source context menu, see 5.4.
Executed command: `variable := value`

Show function result — Display the return value of the just returned function. This is only allowed at a function return point.
Executed command: `?? (.RESULT)`

Assign function result... — Set the value to be returned by the just returned function. This is only allowed at a function return point.
Executed command: `.SET_RESULT value`

Copyright by EADS - All Rights Reserved

5.2.5 The *Call Stack* Menu



First Move to the uppermost level in the subprogram call stack.
Executed command: `.STACK FIRST`

Up Move one level up in the subprogram call stack. Alternatively, you can use the execution control bar, see 5.3.
Executed command: `.STACK UP`

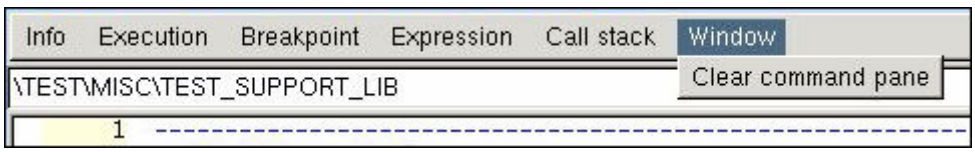
Down Move one level down in the subprogram call stack. Alternatively, you can use the execution control bar, see 5.3.
Executed command: `.STACK DOWN`

Last Move to the lowest level in the subprogram call stack. This is a convenient function to get back to the current execution position, when that position is not currently displayed in the source subwindow. Alternatively, you can use the execution control bar, see 5.3.
Executed command: `.STACK LAST`

Current Same as *Last*. Get back to the current execution position. Alternatively, you can use the execution control bar, see 5.3.
Executed command: `.STACK LAST`

Show List the subprogram call stack.
Executed command: `.STACK`

5.2.6 The *Window* Menu

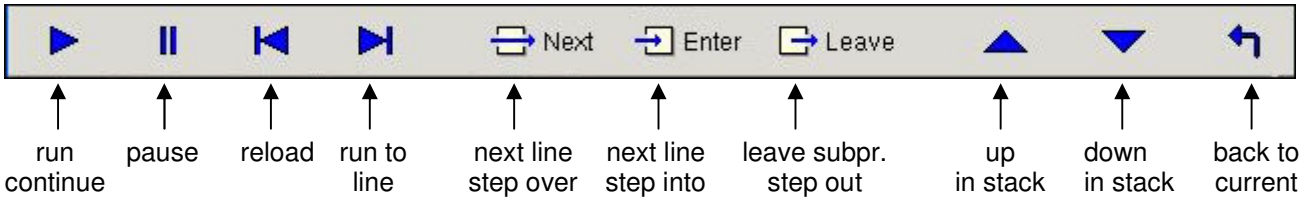


Clear command pane Make the command subwindow empty, leave just one prompt.

Copyright by EADS - All Rights Reserved

5.3 The Execution Control Bar

This bar contains the the most frequently used functions for execution control. All these functions are also available through the menu bar, see 5.2, but for convenience they have been put in a separate and easy-to-access tool bar, as well.




Start or continue execution up to the next breakpoint. Alternatively, you can use the *Execution* menu, see 5.2.2.
Executed command: `.RUN`


Reload the current AP with its original parameter list. The AP is now in initial state, and execution may be started from the beginning.
Executed command: `.LOAD AP_name (parameters)`


Reload the current AP with its original parameter list. The AP is now in initial state, and execution may be started from the beginning. Alternatively, you can use the *Execution* menu, see 5.2.2.
Executed command: `.LOAD AP_name (parameters)`


Continue execution up to the marked line. In order to mark the line, mark at least one character in the line. Alternatively, you can use the *Execution* menu, see 5.2.2, or the context menus, see 5.4 + 5.5.
Executed command: `.RUN line_number`


Execute one source line, step over subprograms, i. e. if the line contains a procedure or function call, do not stop in the subprogram, but execute it completely as part of the line. Alternatively, you can use the *Execution* menu, see 5.2.2.
Executed command: `.NEXT`


Execute one source line, step into subprograms, i. e. if the line contains a procedure or function call, stop on the first executable instruction within the subprogram. Alternatively, you can use the *Execution* menu, see 5.2.2.
Executed command: `.ENTER`


Leave the current subprogram, i. e. continue execution until the subprogram returns to the caller and stop after the call. Alternatively, you can use the *Execution* menu, see 5.2.2.
Executed command: `.LEAVE`


Move one level up in the subprogram call stack. Alternatively, you can use the *Call stack* menu, see 5.2.5.
Executed command: `.STACK UP`

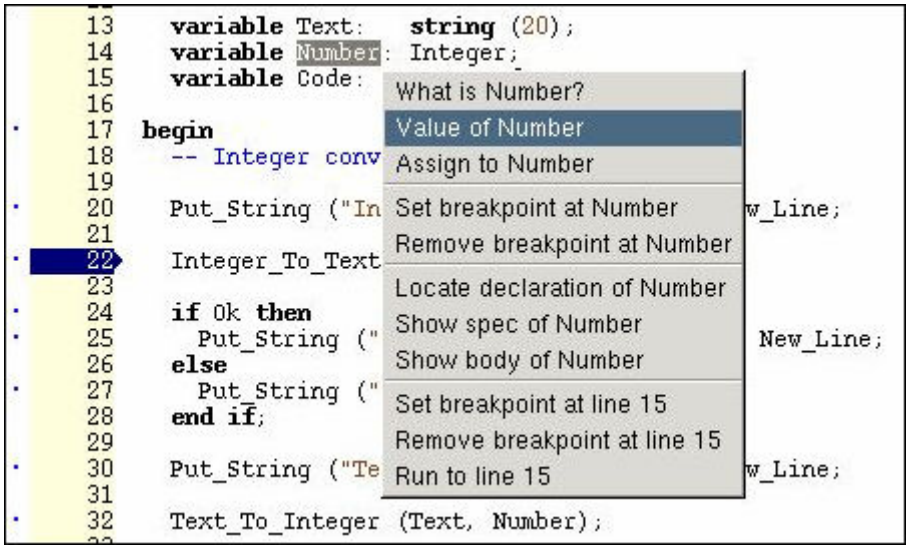

Move one level down in the subprogram call stack. Alternatively, you can use the *Call stack* menu, see 5.2.5.
Executed command: `.STACK DOWN`


Move to the lowest level in the subprogram call stack. This is a convenient function to get back to the current execution position, when that position is not currently displayed in the source subwindow. Alternatively, you can use the *Call stack* menu, see 5.2.5.
Executed command: `.STACK LAST`

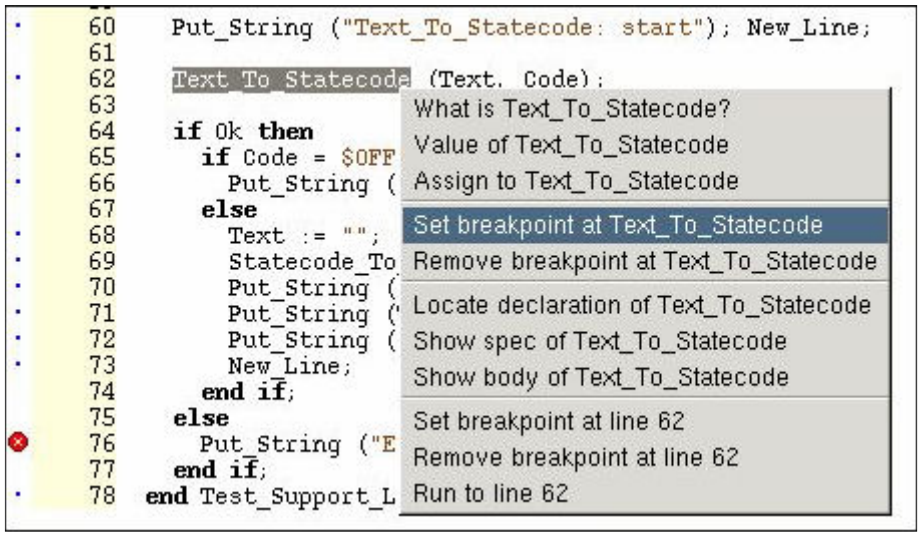
Copyright by EADS - All Rights Reserved

5.4 The Source Context Menu

The source context menu appears when you right-click in the source subwindow. If a portion of text is marked, the marked text will be taken as a parameter to the operation. Otherwise the the debugger will determine the text portion to serve as a parameter from around the right-click position.



show the value of a marked expression



set a breakpoint on a subprogram

What is ...? — Display a short information of what the marked or clicked item is.
Executed command: `?? item`

Value of ... — Display the value of the marked or clicked expression. The expression may be a single item or a complex expression, including software variables, but it must not contain function calls. Alternatively, you can use the *Expression* menu, see 5.2.4.
Executed command: `?? (expression)`

Assign to ... — Assign a value to the marked or clicked variable or software variable. A pop-up dialog will prompt you for the value to be assigned. Alternatively, you can use the *Expression* menu, see 5.2.4.
Executed command: `variable := value`

Copyright by EADS - All Rights Reserved

Set breakpoint at ... — Set a breakpoint on the marked or clicked procedure or function.
Executed command: `.BREAKPOINT subprogram`

Remove breakpoint at ... — Remove the breakpoint from the marked or clicked procedure or function.
Executed command: `.REMOVE subprogram`

Locate declaration of ... — Find and show the source code position where the marked or clicked item is declared. If it is declared in a different compilation unit, the source text of that unit will be displayed in the source subwindow.
Executed command: `.LOCATE item`

Show spec of ... — Display the specification of the marked or clicked library in the source subwindow.
Executed command: `.SHOW library, SPECIFICATION : TRUE`

Show body of ... — Display the body of the marked or clicked library in the source subwindow.
Executed command: `.SHOW library`

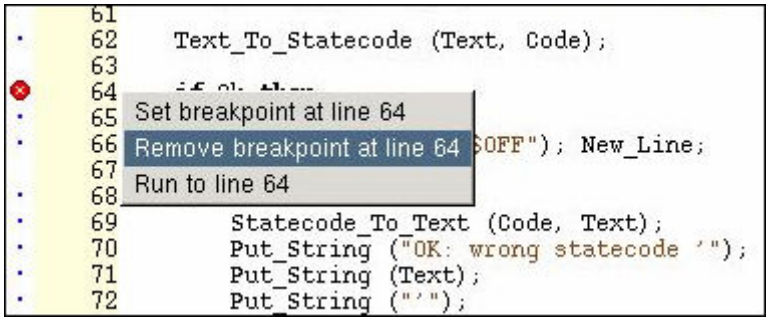
Set breakpoint at line ... — Set a breakpoint in the marked or clicked line. If a portion of text is marked, the breakpoint will be set in the marked line, otherwise in the line you right-clicked with the mouse.
Executed command: `.BREAKPOINT line_number`

Remove breakpoint at line ... — Remove the breakpoint from the marked or clicked line. If a portion of text is marked, the breakpoint will removed set in the marked line, otherwise from the line you right-clicked with the mouse.
Executed command: `.REMOVE line_number`

Run to line ... — Continue execution up to the marked or clicked line. If a portion of text is marked, execution continues up to the marked line, otherwise up to the line you right-clicked with the mouse.
Executed command: `.RUN line_number`

5.5 The Line Context Menu

The line context menu appears when you right-click in the line number column.



Set breakpoint at line ... — Set a breakpoint in the clicked line.
Executed command: `.BREAKPOINT line_number`

Remove breakpoint at line ... — Remove the breakpoint from the clicked line.
Executed command: `.REMOVE line_number`

Run to line ... — Continue execution up to the clicked line.
Executed command: `.RUN line_number`

Copyright by EADS - All Rights Reserved

6. Specific Debugger Functionality

6.1 Name Scopes and Visibility

All named objects, such as types, variables, constants and subprograms, belong to some name *scope*. During a debugging session different name scopes will exist, new scopes may be opened and open scopes may be closed. In order to understand which identifier is visible at what time, an understanding of the scoping behaviour of the debugger is indispensible.

Name scopes make up a stack, according to the scoping rules of UCL. So the same scope stack that was in effect when the envolved compilation units were compiled will be found when debugging through these compilation units. But in the debugger, additional scopes play a role: The command subwindow holds its own HLCL session scope where you may declare named objects and import libraries that are not involved in the unit being debugged. And when you execute command sequences, those sequences may have a local scope with local declarations and local imports.

The debugger maintains a *current scope*, wich represents the "standpoint" of the user, from where he views the different name scopes and wich, thus, determines which items he can directly see and which object a particular identifier actually denotes. Different user actions may change the current scope, thereby moving the user's standpoint into a different scope:

- Basically, the current scope is determined by the current execution position. When, e. g., executing a line within a local procedure of the AP, the current scope (and the standpoint of the user) is the local scope of the procedure. From here, he first sees the local declarations and parameters, then the global declarations and parameters on AP level and finally the imported objects. If the procedure is in an imported library, the user will first see the local declarations and parameters of the procedure, then the global declarations within the library, and finally the objects imported by the library. When execution resumes, a return from the procedure will switch the current scope to the scope of the caller, either the AP or library body or another subprogram, possibly in a different library.
- When navigating through the subprogram call stack (e. g. with the up and down buttons), the current scope will switch to the scope of the caller or the callee, respectively. A click on the *back* button brings it back to the current execution position.
- When selecting a different compilation unit from the module list and thereby displaying that compilation unit in the source subwindow, the current scope switches to the body of the selected module. So when, e. g. selecting a library, the user will see the global declarations of the library (specification + body) and then the objects imported in the library.

Ony after the complete scope stack of the debugged AP the user will see the HLCL session scope maintained in the command window. And when a command sequence is being executed, its local scope and local imports will be seen before all other scopes. So the overall scope (and visibility) stack within a debugging session is, in decreasing order:

1	2	3	4
the currently running HLCL command sequence 1. `for` loop stack (innermost loop first) 2. parameters + local declarations 3. imports	**the UCL scope stack** 1. local subprogram: a) `for` loop stack b) parameters + local declarations 2. AP/library body: a) `for` loop stack b) AP parameters + global declarations (spec. + body) 3. imports	**the HLCL session scope** 1. session declarations 2. imports	**predefined identifiers** 1. application 2. HLCL 3. UCL

Copyright by EADS - All Rights Reserved

How to Handle Name Conflicts?

When objects with the same name (identifier) exist in different scopes, the identifier in the highest scope will hide objects with the same identifier in lower scopes. This may easily cause name conflicts between the debugged UCL module and the HLCL session and command sequences:

- An identifier locally declared in a command sequence hides an identifier in the UCL module being debugged. To avoid this situation, choose names in your sequence that will most likely not conflict with identifiers in the debugged unit, e. g. give them a specific prefix.
- An identifier in the HLCL session is hidden by an identifier in the debugged module. Here the same solution applies: choose names in your HLCL session that will most likely not conflict with names in the debugged module.
- An identifier in the debugged module hides a predefined identifier, e. g. a `Put` procedure in the AP hides the predefined HLCL `Put` command. This problem can easily be avoided by using predefined identifiers in qualified notation with a leading dot (`.Put`, `.Max`, `.Length`).

Other name conflicts that are common for any HLCL commanding, such as between command sequences and an HLCL session are not mentioned here.

6.2 Assignments

The UCL Debugger allows you to assign values not only to variables and software variables, but also to objects that do normally not accept assignments:

- **`for`** loop variables
- **`in`** parameters (parameters of mode **`in`**)

While assignments to **`for`** loop variables do not present a particular problem (the loop will just terminate sooner or later), assignments to **`in`** parameters require uttermost care:

- Parameters of scalar types are passed by copy, so an assignment will just overwrite the local copy within the subprogram. Structured parameters, on the other hand, are passed by reference, an assignment will therefore overwrite the actual parameter outside the subprogram.
- If the actual parameter is a constant, an assignment may have very odd effects:
  - Constants are kept in a separate area of the stack machine memory, the *constant table*. An assignment will, thus, change the constant value, and the same constant may suddenly appear as a different value in other parts of the program.
  - For optimization, the UCL compiler attempts to share identical parts of the constant table between several constants. So altering one constant through assignment to an **`in`** parameter may implicitly change (parts of) other constants, as well.
  - For string constants, only the smallest possible portion is actually allocated in the constant area, i. e. a string whose maximum length is equal to the actual length, even if the string type has a much larger maximum length. For an empty string only one word containing the actual length (= 0) will be allocated. So when a string constant is passed as a parameter whose type is a fixed length string, the actually allocated memory may (and will normally) be smaller than given by the string type. An assignment to this parameter will be checked against the formal maximum size, and the assignment may overwrite not only the passed constant, but also constants allocated at adjacent positions in the constant table.

Copyright by EADS - All Rights Reserved

7. The HLCL Command Subwindow

The HLCL command subwindow allows you to control the debugger on textual command level. Not only the specific debugging funtionality is available here, but also all usual HLCL features, with the only exception that you cannot start APs and call library subprograms. In particular, the user may automat parts or all of a debugging session through HLCL command sequences.

The command subwindow implements a simple command history. You can navigate through the history by pressing the up and down arrow keys. The left and right arrow keys move the cursur within the command line which may be edited and reentered by pressing the return key.

The following subsections describe the specific debugger commands and functions, as well as the use of command sequences for debugging. For general HLCL commanding see the HLCL Reference Manual [2.2.3].

7.1 Specific Debugger Commands and Functions

The UCL Debugger predefines some debugger specific commands and functions.

Commands:

LOAD	load or reload a module into the debugger
RUN	start or continue execution
NEXT	execute one line, step over subprogram calls
ENTER	execute one line, step into subprogram calls
LEAVE	execute up to the return from the current subprogram
BREAKPOINT	set/list breakpoint(s)
REMOVE	remove breakpoint(s)
LOCATE	locate declaration of item
SET_RESULT	set function result being returned
SHOW	show source at given position
STACK	move in the subprogram call stack
I_CODE	show executed I-Code instructions

Functions

AT_BREAKPOINT	number of current breakpoint
AT_LINE	current line number
AT_MODULE	current module pathname
NO_OF_BREAKPOINT	number of last breakpoint set
RESULT	function result being returned

Furthermore, the UCL Debugger predefines a few types used for the parameters of some commands:

Types

```
type AP_PATHNAME    = pathname (UCL_AUTOMATED_PROCEDURE)
type LEVEL          = UNSIGNED_INTEGER
type LINE_NUMBER    = UNSIGNED_INTEGER
type STACK_COMMAND  = (UP, DOWN, FIRST, LAST)
type SUBPROGRAM     = entity (procedure, function)
```


Copyright by EADS - All Rights Reserved

7.1.1 Debugger Commands

command BREAKPOINT ...
  in POSITION : **union** (SUBPROGRAM, LINE_NUMBER) := **null**

When given without a parameter, all currently set breakpoints are listed. With a parameter, set a breakpoint at the given position. The position can be given as a line number in the currently displayed source text, or the (qualified or single) name of a procedure or function. In the latter case, execution will stop at the first code position in the subprogram when it is called.

BREAKPOINT	list all breakpoints currently set
BREAKPOINT 185	set breakpoint in line 185
BREAKPOINT User_Lib.Proc	set breakpoint in procedure Proc in User_Lib

command ENTER

Execute one source line. If the line contains a procedure or function call, enter the subprogram and stop at the first executable line.

command I_CODE ...
  in ON : BOOLEAN := **null**

Switch output of I-Code on or off. When on, the executed I-Code instructions will be displayed for each executed source line.

I_CODE True	switch I-Code output on
I_CODE False	switch I-Code output off

command LEAVE

Leave the current subprogram, i. e. execute all instructions up to, and including, the return from the subprogram. For a function, you may now obtain or change the function return value, using the RESULT function or the SET_RESULT command.

command LOAD ...
  in MODULE : AP_PATHNAME () := **null**

Load an AP into the debugger. The AP is to be given together with its parameters in parentheses. If an AP is already loaded, you can reload this AP, either with different parameters (if you give a parameter list) or with the same parameters (if you omit the parameter list). It is not possible to load a different AP, if an AP is already loaded.

LOAD *some**path*\AP (*parameters*)	load or reload an AP
LOAD *some**path*\AP	reload same AP with same parameters

command LOCATE ...
  in ITEM : **entity**

Find and show the source code position where a named item is declared. If it is declared in a different compilation unit, the source text of that unit will be displayed in the source subwindow.

command NEXT

Execute one source line. If the line contains a procedure or function call, do not enter the subprogram, but execute it.

Copyright by EADS - All Rights Reserved

```
command REMOVE ...
  in BREAKPOINT : union (SUBPROGRAM, LINE_NUMBER)
```

Remove a breakpoint from a line in the current compilation unit, or from a subprogram.

`REMOVE 185`	remove breakpoint from line 185 (0 = all breakpoints)
`REMOVE User_Lib.Proc`	remove breakpoint from procedure `Proc` in `User_Lib`
`REMOVE 0`	remove all breakpoints

```
command RUN ...
  in POSITION : union (SUBPROGRAM, LINE_NUMBER) := null
```

Start or continue execution. If a position parameter is given, execution will stop at that position.

`RUN`	start or continue execution up to next breakpoint
`RUN 185`	start or continue execution up to line 185
`RUN User_Lib.Proc`	start or continue execution up to procedure `Proc` in `User_Lib`

```
command SET_RESULT ...
  in VALUE : union
```

This command can only be given at a function return point. It sets the function return value to the given value. The type of the `VALUE` parameter changes according to the function result type.

`SET_RESULT $OFF`	set the return value of the just returned function to `$OFF`.

```
command SHOW ...
  in POSITION      : union (entity (pathname), SUBPROGRAM, LINE_NUMBER)
  in SPECIFICATION : BOOLEAN := FALSE
```

Display the given source text position in the source subwindow. If the position is in a different compilation unit, that unit will be displayed in the source subwindow. The `SPECIFICATION` parameter can be used to require the body or specification of a library to be displayed.

`SHOW 225`	move display to line 225 of the currently dislayed compilation unit
`SHOW Put_Value`	move display to procedure `Put_Value`
`SHOW \...\TIME_LIBRARY`	show body of library `\...\TIME_LIBRARY` in source subwindow
`SHOW \...\TIME_LIBRARY, SPECIFICATION : TRUE`	same, but show specification of the library

```
command STACK ...
  in LEVEL : union (STACK_COMMAND, LEVEL) := null
```

List or navigate within the subprogram call stack:

`STACK`	list the call stack
`STACK UP`	go one level up in the call stack
`STACK DOWN`	go one level down in the call stack
`STACK FIRST`	goto the top of the call stack
`STACK LAST`	goto the bottom of the call stack (current execution position)
`STACK 3`	goto level 3 of the call stack

Copyright by EADS - All Rights Reserved

7.1.2 Debugger Functions

function AT_BREAKPOINT : UNSIGNED_INTEGER

When stopped at a breakpoint, this function returns the number of this breakpoint.

function AT_LINE : LINE_NUMBER

Returns the current line number.

function AT_MODULE : **pathname**

Returns the pathname of the module currently executed.

function NO_OF_BREAKPOINT : UNSIGNED_INTEGER

Returns the number of the breakpoint just set. This can be used to store breakpoint numbers for later reference.

```
BREAKPOINT Put_Value                    set a breakpoint
Put_Value_BP := NO_OF_BREAKPOINT        keep its number in a variable
...
if AT_BREAKPOINT = Put_Value_BP then    reference the stored breakpoint number
   ...
end if;
```

function RESULT : **union**

This function is only allowed at a function return point. It yields the value returned by the function. The command SET_RESULT may be used to have the function return a different value.

Copyright by EADS - All Rights Reserved

7.2 HLCL Command Sequences

You may automat parts or all of a debugging session by using HLCL command sequences. Within the sequences, the usual HLCL language features are available, in particular the specific debugger commands and functions, see 7.1.

Please note that command procedures (closed sequences) cannot reasonably be used when access to named objects in the debugged unit are needed, since command procedures do not allow any visibility outside their local scope. Command sequences used for debugging will, therefore, mostly be open sequences.

Here is a simple example for the usage of a command sequence. Suppose you want to debug a **`for`** loop in an AP like the following up from the n-th cycle, i. e. ignore n - 1 cycles and then break for manual debugging:

```
  1: procedure Some_AP;
        ...
 80: begin
        ...
105:    for I := 1 to 1000 do
106:       ...   -- stop here in 70-th iteration cycle
           ...
           ...
178:    end for;
        ...
379: end Some_AP;
```

Then the following sequence may be used to interrupt any such loop in a specific line at a specific iteration cycle. The loop index, the line number and the cycle number are passed as parameters:

```
sequence Stop_Loop (in out Index : Integer;
                    in     Line  : Unsigned_Integer;
                    in     Cycle : Integer);

   variable BP : Unsigned_Integer;  -- breakpoint number

begin
   .BREAKPOINT Line;                        -- set breakpoint in line
   BP := .NO_OF_BREAKPOINT;                 -- keep the breakpoint number

   loop
      .RUN;                                 -- start/continue execution

      if .AT_BREAKPOINT = BP then           -- we are at our breakpoint
         if Index = Cycle then              -- desired iteration cycle reached
            .Put "Stopped in line " + string (Line);
            exit;
         end if;
      elsif .AT_BREAKPOINT = 0 then         -- we are at end of program
         .Put "Loop ended prematurely, stopped at end of program";
         exit;
      else
         -- nothing, this breakpoint is not ours
      end if;
   end loop;
end Stop_Loop;
```

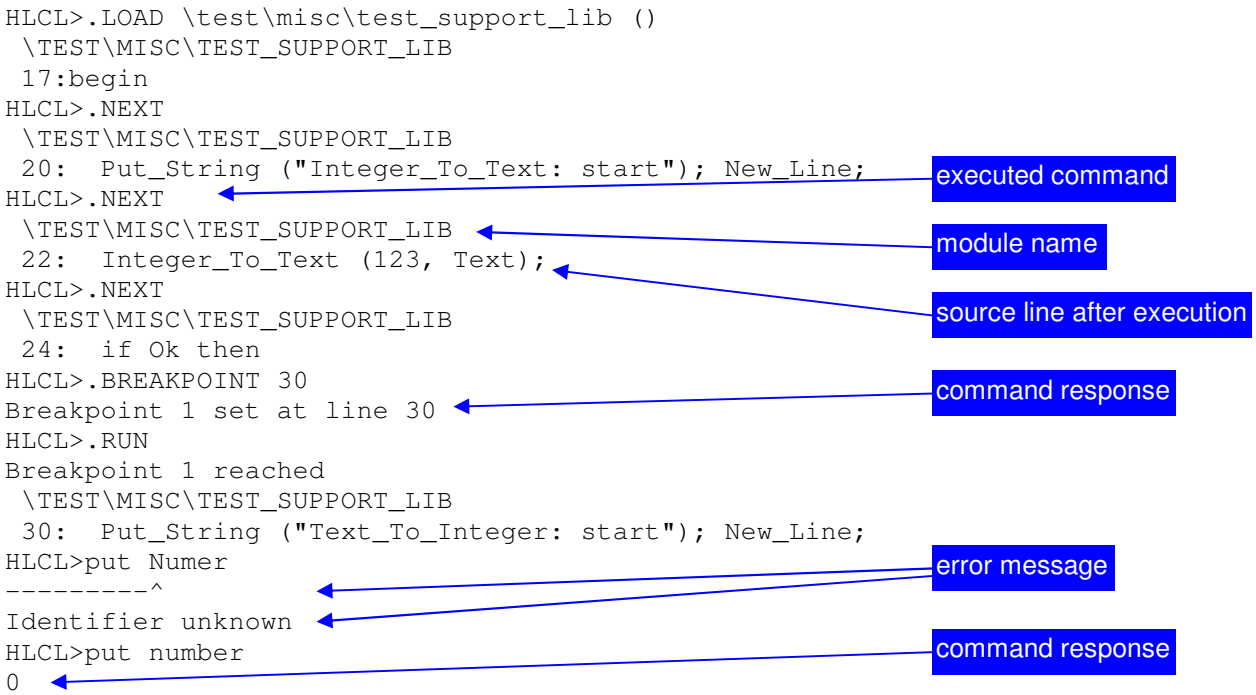
For the above AP, call the sequence as follows:

```
Stop_Loop I, 106, 70
```


Copyright by EADS - All Rights Reserved

8. The Log File

The UCL Debugger allows to create a log file and log all user interactions, together with debugger responses and the source line, that will be executed next. The log file has the following format:



- Lines starting with the command prompt "`HLCL>`" contain the executed commands.
- Source lines hava a line number prefix of the form "`30:` ".
- Each source line is preceded by a line with the name of the module (compilation unit) it belongs to.
- Command responses and error messages follow without any specific marker.

Copyright by EADS - All Rights Reserved