
PROLOG

reference manual

ARM Evaluation System

Acorn OEM Products



PROLOG

Part No 0448,012
Issue No 1.0
14 August 1986

© Copyright Acorn Computers Limited 1986

Neither the whole nor any part of the information contained in, or the product described in, this manual may be adapted or reproduced in any material form except with the prior written permission of the copyright holder. The only exceptions are as provided for by the Copyright (photocopying) Act, or for the purpose of review, or in order for the software herein to be entered into a computer for the sole use of the owner of this book.

Within this publication the term 'BBC' is used as an abbreviation for 'British Broadcasting Corporation'.

- The manual is provided on an 'as is' basis except for warranties described in the software licence agreement if provided.
- The software and this manual are protected by Trade secret and Copyright laws.

The product described in this manual is subject to continuous developments and improvements. All particulars of the product and its use (including the information in this manual) are given by Acorn Computers in good faith.

There are no warranties implied or expressed including but not limited to implied warranties or merchantability or fitness for purpose and all such warranties are expressly and specifically disclaimed.

In case of difficulty please contact your supplier. Every step is taken to ensure that the quality of software and documentation is as high as possible. However, it should be noted that software cannot be written to be completely free of errors. To help Acorn rectify future versions, suspected deficiencies in software and documentation, unless notified otherwise, should be notified in writing to the following address:

Customer Services Department,
Acorn Computers Limited,
645 Newmarket Road,
Cambridge
CB5 8PD

All maintenance and service on the product must be carried out by Acorn Computers. Acorn Computers can accept no liability whatsoever for any loss, indirect or consequential damages, even if Acorn has been advised of the possibility of such damage or even if caused by service or maintenance by unauthorised personnel. This manual is intended only to assist the reader in the use of the product, and therefore Acorn Computers shall not be liable for any loss or damage whatsoever arising from the use of any information or particulars in, or any error or omission in, this manual, or any incorrect use of the product.

Econet® and The Tube® are registered trademarks of Acorn Computers Limited.

ISBN 1 85250 007

Published by:

Acorn Computers Limited, Fulbourn Road, Cherry Hinton, Cambridge CB1 4JN, UK

Edited by William Clocksin

Based on material from the C-PROLOG User's Manual by Fernando Pereira, which is based on the EMAS PROLOG User's Manual by Luis Dama which in turn is based on the User's Guide to DEC system- 10/ 20 PROLOG by Fernando Pereira, David Warren, David Bowen, Lawrence Byrd, and Luis Pereira.

Produced by Baddeley Associates Limited, Cambridge

Contents

1. About this guide	1
1.1 Conventions used in this guide	1
2. About PROLOG	2
3. Using ARM PROLOG	3
3.1 Access to ARM PROLOG	3
3.2 Reading in programs	4
3.3 Questions	5
3.4 Saving a program state	7
3.5 Restoring a saved program	8
3.6 Program execution and interruption	8
3.7 Nested executions: break and abort	9
3.8 Leaving PROLOG	10
4. Summary of PROLOG syntax	11
1 Comments	11
2 Terms	11
3 Variables	12
4 Compound terms	12
5 Operators	15
4.1 Syntax errors	20
5. Summary of PROLOG semantics	21
5.1 Declarative and procedural semantics	24
5.2 Occurs check	26
5.3 Specifying control information	27
6. ARM PROLOG's built-in predicates	29
6.1 Input/ output	30
6.2 Reading in programs	31
6.2.1 compile(F)	31
6.2.2 consult(F)	31
6.2.3 [F1, F2, ..., Fn]	31
6.3 Opening and closing files	31
6.3.1 see(F)	31
6.3.2 seeing(F)	31
6.3.3 seen	31
6.3.4 tell(F)	31
6.3.5 telling(F)	32
6.3.6 told	32

6.4 Reading and writing PROLOG terms	32
6.4.1 read(X)	32
6.4.2 read(X,Y)	32
6.4.3 write(X)	32
6.4.4 display(X)	32
6.4.5 writeq(X)	32
6.5 Getting and putting characters	33
6.5.1 nl	33
6.5.2 get0(N)	33
6.5.3 get(N)	33
6.5.4 skip(N)	33
6.5.5 put(N)	33
6.5.6 tab(N)	33
6.6 Arithmetic	33
6.7 Affecting the flow of the execution	35
6.7.1 , Q	35
6.7.2 P ; Q	35
6.7.3 true	35
6.7.4 X = Y	35
6.7.5 X \ = Y	35
6.7.6 !	36
6.7.7 \ +P	36
6.7.8 P -> Q ; R	36
6.7.9 P -> Q	36
6.7.10 repeat	36
6.7.11 fail	36
6.7.12 forall(G,T)	36
6.8 Classifying and operating on PROLOG terms	37
6.8.1 var(X)	37
6.8.2 nonvar(X)	37
6.8.3 atom(X)	37
6.8.4 integer(X)	37
6.8.5 atomic(X)	37
6.8.6 functor(T,F,N)	37
6.8.7 arg(I,F,X)	37
6.8.8 X =.. Y	38
6.8.9 name(X,L)	38
6.8.10 call(X)	38
6.8.11 X (where X is a variable)	39
6.8.12 numbervars(T,M,N)	39

6.9 Processing sets	39
6.9.1 setof(X,P,S)	39
6.9.2 bagof(X,P,B)	40
6.9.3 findall(X,P,L)	41
6.9.4 Y^Q	41
6.10 Comparing terms	41
6.10.1 X == Y	42
6.10.2 X \ == Y	42
6.10.3 T1@<T2	42
6.10.4 T1@>T2	42
6.10.5 T1@=<T2	42
6.10.6 T1@>=T2	42
6.10.7 compare(Op, T1, T2)	42
6.10.8 sort(L1, L2)	43
6.10.9 keysort(L1, L2)	43
6.11 Manipulating the PROLOG program database	43
6.11.1 assert(C)	43
6.11.2 assert(C,R)	44
6.11.3 asserta(C)	44
6.11.4 asserta(C,R)	44
6.11.5 assertz(C)	44
6.11.6 assertz(C,R)	44
6.11.7 clause(P,Q)	44
6.11.8 clause(P,Q,R)	44
6.11.9 retract(C)	45
6.11.10 abolish(N,A)	45
6.11.11 listing(N,A)	45
6.12 Manipulating the internal indexed database	45
6.12.1 recorded(K,T,R)	46
6.12.2 recorda(K,T,R)	46
6.12.3 recordz(K,T,R)	46
6.12.4 erase(R)	46
6.13 Interacting with the programming environment	46
6.13.1 writedepth(X,Y)	46
6.13.2 writewidth(X,Y)	47
6.13.3 unknown(X,Y)	47
6.13.4 op(P,T,N)	47
6.13.5 break_handler	47
6.13.6 error_handler(N,X)	48
6.13.7 abort	48

6.13.8	save(F)	48
6.13.9	statistics(X,Y)	48
6.13.10	system(X)	48
6.14	Defining modules	49
6.14.1	module(X)	50
6.14.2	endmodule(X)	50
6.14.3	visa(A,F)	50
6.14.4	sacred	50
6.14.5	omni	51
6.14.6	import(F,M)	51
7.	What's particular to ARM PROLOG	53
7.1	The user language	53
7.2	Implementation details	54
7.3	Restrictions on data structures	54
7.3.1	Integers	54
7.3.2	Strings	55
7.3.3	Atoms	55
7.3.4	Variables	55
7.3.5	Compound terms	55
7.4	Errors	55
7.5	Input and output	56
7.6	Operator declarations	56
7.7	Built-in predicates exported to the user	57
7.8	Arithmetic expressions	60



1. About this guide

If PROLOG is new to you, you should first read a standard text such as *Programming in PROLOG* (by W F Clocksin and C S Mellish, published by Springer-Verlag, 1981).

This guide includes:

- instructions on starting and running a basic PROLOG session on the ARM evaluation system (starting on page 3)
- two chapters which together give a quick-reference summary of the main points covered in *Programming in PROLOG* (pages 11 and 21)
- a detailed guide to the built-in predicates available in ARM PROLOG (page 29)
- a summary of the ways in which ARM PROLOG differs from other versions of PROLOG (page 53).

1.1 Conventions used in this guide

As you work with the PROLOG interpreter, you type a full stop then press

RETURN

to indicate the end of a line. PROLOG then interprets what you have typed. For simplicity, in this guide we do not show the **RETURN**, and we show the full stop only in:

- syntax definitions
- full examples.

2. About PROLOG

PROLOG is a simple but powerful programming language originally developed at the University of Marseilles as a practical tool for programming in logic. PROLOG is especially suitable for high-level symbolic programming tasks and has been applied in many areas of artificial intelligence research.

The interactive ARM PROLOG system consists of an incremental PROLOG compiler, a run-time system, and a wide range of built-in (system-defined) procedures. The system is closely compatible with DECsystem-10 PROLOG and thus is reasonably compatible with descendants of DECsystem-10 PROLOG, such as C-PROLOG, Quintus PROLOG, and POPLOG.

3. Using ARM PROLOG

The text of a PROLOG program is normally created in a number of files using the TWIN editor. ARM PROLOG can then be instructed to read in programs from these files. ARM PROLOG can read in programs in either of two ways:

- consulting
- compiling.

When a file is consulted, it is read in together with information about the clause needed when debugging the program. During normal use, however, such extra information is not required, and programs are usually compiled. Consulting takes more time than compiling, and consulted programs occupy more store than compiled ones.

It is recommended that you make use of a number of different files when writing programs. Since you will be editing and consulting individual files, it is useful to use files to group together related procedures, keeping collections of procedures that do different things in different files. Thus a PROLOG program will consist of a number of files, each file containing a number of related procedures.

When your programs start to grow to a fair size, it is also a good idea to have one file which only contains commands to the system to consult all the other files which form a program. You will then be able to consult your entire program by just consulting this single file.

3.1 Access to ARM PROLOG

Because PROLOG makes syntactic use of the difference between upper-case and lower-case letters it is important that you have your keyboard set up so that it accepts lower case in the normal way. Hence, ensure the CAPS LOCK and SHIFT LOCK lamps are not lit.

To enter PROLOG, type:

```
prolog
```

PROLOG will output a banner and prompt you for directives as follows:

```
ARM PROLOG Version 1.23
?-
```

There will be a pause before the banner and the prompt while the system loads itself. It is possible to type ahead during this period if you get impatient.

Once you are in PROLOG, there are three important keys or sequences to remember:

- **(RETURN)** terminates an input line
- **(CTRL)D** marks end of input
- **(ESCAPE)** interrupts execution of a program.

3.2 Reading in programs

A program is made up of a sequence of clauses, possibly interspersed with directives to the compiler. The clauses of a procedure do not have to be immediately consecutive, but remember that their relative order in the file controls the order in which the system tries them.

To input a program from a file *file*, give the directive:

```
?- [file].
```

which will instruct the system to consult the program. The file specification *file* must be a PROLOG atom. It may be any ADFS filename specification. PROLOG does not use special extension. Note that if this filename contains characters which are not normally allowed in an atom, then it is necessary to surround the whole file specification with single quotes (since quoted atoms can include any character). For example:

```
?- ['joe.test'].
```

The specified file is then read in. Clauses in the file are stored in the database ready to be executed, while any directives are obeyed as they are encountered.

In general, this directive can be any list of filenames, such as:

```
?- [myprogram, extras, testbits].
```

In this case all three files would be consulted.

To compile clauses, use the `compile` predicate, which will take either an atom or a list of atoms as an argument:

```
?- compile(prog).
```

```
?- compile([myprogram, 'jon.extras', testbits]).
```

Compilation is the recommended way to read in clauses, as it is faster and the program will occupy less store.

Clauses may also be typed in directly. To enter a clause, you must give the directive:

```
?- [user].
```

```
:
```

The system is now in a state where it expects input of clauses or directives. This is indicated by the `:` prompt. To get back to the top level of the system, type `(CTRL)D`.

Typing clauses directly into ARM PROLOG is only recommended if the clauses will not be needed permanently, and are few in number. For larger programs you should use TWIN to produce a file containing the text of the program.

3.3 Questions

When PROLOG is at top level (signified by an initial prompt of `?`, it reads in terms and treats them as directives to the system to try and satisfy some goals. These directives are called questions (or queries). Remember that PROLOG terms must terminate with a full stop (`.`), and that therefore PROLOG will not execute anything for you until you have typed the full stop, and then pressed `(RETURN)` at the end of the directive.

For example, suppose list membership has been defined by the following (where `_` is an anonymous variable):

```
member(X, [X|_]).
```

```
member(X, [_|L]) :- member(X, L).
```

If the goals specified in a question can be satisfied, and if there are no variables as in this example:

```
?- member(b, [a,b,c]).
```

then the system answers:

```
yes
```

and execution of the question terminates.

If variables are included in the question, then the final value of each variable is displayed, except for anonymous variables. Thus the question:

```
?- member(X, [a,b,c]).
```

would be answered by:

```
X = a  
more (y/ n)?
```

At this point the system waits for you to indicate whether that solution is sufficient, or whether you want it to backtrack to see if there are any more solutions. Typing *n* (for no) followed by **RETURN** terminates the question, while typing *y* (for yes) followed by **RETURN** causes the system to backtrack to look for alternative solutions. If no further solutions can be found it outputs:

```
no
```

The outcome of some questions is shown below.

```
?- member(X, [tom,dick,harry]).
X = tom
more (y/n)? y
X = dick
more (y/n)? y
X = harry
more (y/n)? y
no
?- member(X, [a,b,f(Y,c)]), member(X, [f(b,Z),d]).
X = f(b,c)
Y = b
Z = c
more (y/n)? n
yes
?- member(X, [f(_),g]).
X = f(_1728)
more (y/n)? y
X = g
more (y/n)? y
no
?-
```

When PROLOG reads terms from a file (or from the keyboard following a call to `[user()]`), it treats them all as program clauses. In order to get the system to execute directives from a file they must be preceded by `?-`.

3.4 Saving a program state

Once a program has been read, the interpreter will have available all the information necessary for its execution. This information is called a program state.

The state of a program may be saved on a file for future execution. To save a program into a file `file`, call the goal:

```
?- save(file).
```

The goal `save` can be called at top level, from within a break level (see below), or from anywhere within a program.

Note that `save` only makes a copy of the current memory contents. Saving a state does not make any changes to source files. If a new version of PROLOG is installed, saved states created with the old version will almost certainly become unusable. You are thus advised to keep up to date source files for your programs at all times, and only use saved states for convenience.

3.5 Restoring a saved program

Once a program has been saved into a file `file`, PROLOG can be restored to this saved state by calling the goal:

```
?- restore(file).
```

After execution of this goal, the interpreter will be in exactly the same state as existed immediately prior to the call to `save`, except for open files, which are automatically closed by `save`. That is to say, execution will start at the goal immediately following the call to `save`, just as if `save` had succeeded. If you saved the state at top level then you will be back at top level, but if you explicitly called `save` from within your program then the execution of your program will continue.

There is another version of `save`:

```
save(file, Restart)
```

will save into the file `file` as before, and will instantiate the variable `Restart` to 0. When the file is restored, `Restart` will be instantiated to 1.

3.6 Program execution and interruption

Execution of a program is started by giving the system a directive which contains a call to one of the program's procedures.

Only when execution of one directive is complete does the system become ready for another directive. However, one may interrupt the normal execution of a directive by pressing `ESCAPE` causes the goal `break_handler` to be called at the earliest opportunity.

3.7 Nested executions: break and abort

PROLOG provides a way to suspend the execution of your program and to enter a new incarnation of the top level where you can issue directives to solve goals. When the built-in predicate `break_handler` is called, the message:

```
(Break) ?-
```

will be displayed. This signals the start of a break-level and except for the effect of `abort` (see below), it is as if the interpreter were at top level. If `break_handler` is called within a break-level, then another recursive break-level is started. Break-levels may be nested arbitrarily deeply, although this practice is not very useful.

Typing `(CTRL) D` will close the break-level and resume the execution which was suspended, starting at the procedure call where the suspension took place. Do look at the prompt before you type `(CTRL) D`. If the prompt is `?-` you are at the very top level, and `(CTRL) D` will take you right out of PROLOG, and your current state will be irrevocably lost. An equivalent way of getting out of break levels is to type:

```
end_of_file.
```

in response to the `?-` prompt. This too will take you out of PROLOG if you are at the top level. It has the advantage over

`(CTRL) D` that you can easily and clearly include it in scripts, should you want to call `break_handler` in a script.

To abort the current execution, forcing an immediate failure of the directive being executed and a return to the top level of the system, call the built-in predicate `abort`, either from the program or by executing the directive:

```
?- abort.
```

within a break. In this case no `(CTRL) D` is needed to close the break, because all break levels are discarded and the system returns right back to top level.

3.8 Leaving PROLOG

To leave ARM PROLOG, type:

```
?- halt.
```

This directive can be issued either at top level, or within a break-level, or indeed from within your program.

If your program is still executing then you should interrupt it and abort to return to top level so that you can call `halt`.

Typing `CTRL`D at top level also causes ARM PROLOG to terminate.

4. Summary of PROLOG syntax

For full details on PROLOG syntax, see *Programming in PROLOG*.

1. Comments

ARM PROLOG supports two kinds of comments. One form is:

```
/ * comment text */
```

These comments can stretch over any number of lines and pages. That is precisely the problem with them. If you forget a closing comment bracket your comment will quietly eat large quantities of your program. So these comments are not favoured for annotations within clauses. Also, the comment brackets may not be nested. The other style of comment is end of line comments:

```
% comment reaching to the end of the line
```

which are opened by a percent sign and closed by a new line.

2. Terms

The data objects of PROLOG are called terms. A term is either a constant, a variable or a compound term.

The constants include integers such as:

```
0    -999    8'177    2'101101
```

Constants also include atoms such as:

```
a    void    -    :=    'Algol-68'    []
```

The name of an atom may be any sequence of characters not including the ASCII NUL character. It must be put in single quotes, unless it is:

- a sequence of sign characters
- an identifier starting with a lower case letter
- a singleton special character like ! or ;.

If you want a single quote in a name, it must be written twice. Thus the quotes are needed in each of the atoms spelled as:

```
'!!' '%rem' 'has spaces' '1234' 'NotAVariable' 'a-b'  
'has one quote '' in it'
```

As in other programming languages, constants are definite elementary objects.

3. Variables

Variables are distinguished by an initial capital letter or by the initial character `_`, for example:

```
X Value A A1 _3 _result
```

If a variable is only referred to once, it does not need to be named and may be written as an anonymous variable, indicated by the underline character `_`.

A variable should be thought of as standing for some definite but unidentified object. A variable is not a writable storage location as in most programming languages; rather it is a local name for some data object.

It is possible to write variable names entirely in upper case, but solid upper case is far less readable than mixed case. Nor is it necessary to write constant symbols entirely in lower case; `nilTree` is a perfectly good atom name.

4. Compound terms

The structured data objects of the language are the compound terms. A compound term comprises a functor (called the principal functor of the term) and a sequence of one or more terms called arguments. A functor is characterised by its name, which is an atom, and its arity or number of arguments. For example the compound term whose functor is named `point` of arity 3, with arguments `X`, `Y` and `Z`, is written:

```
point (X, Y, Z)
```

An atom is considered to be a functor of arity 0.