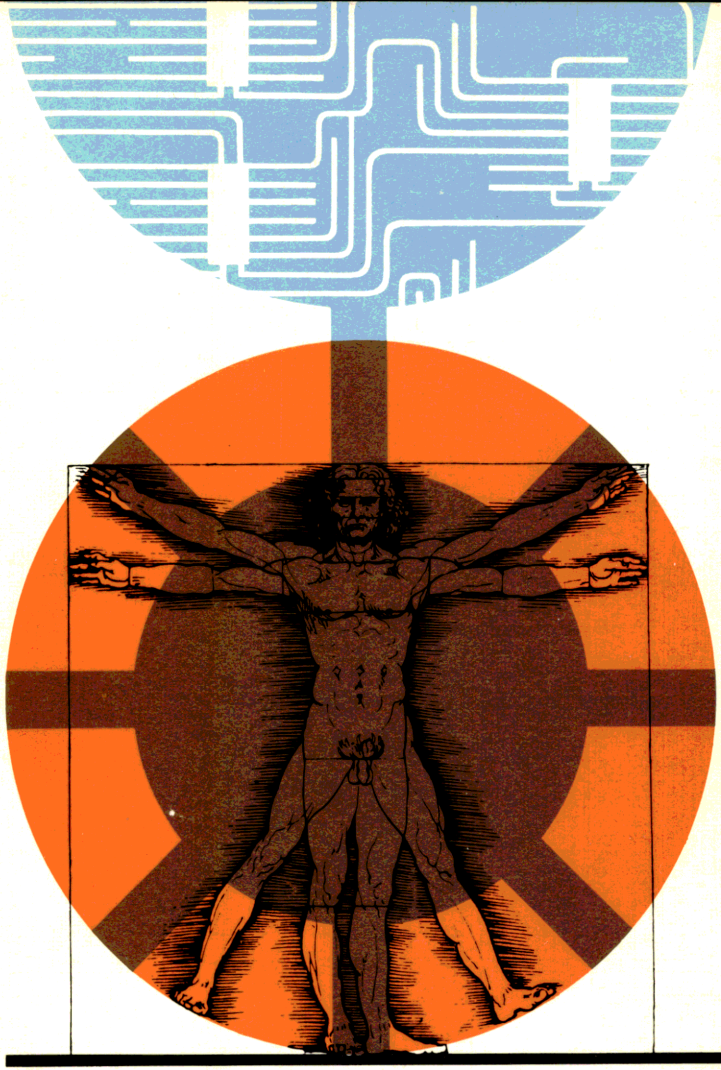


SLANG ASSEMBLER

RC4000®
DATAMATICS

RCSL No: 55-D18

RC 4000 SOFTWARE

SLANG ASSEMBLER

2. Edition

Edited by
Per Brinch Hansen

A/S REGNECENTRALEN
Copenhagen - May 1969

FOREWORD

This manual is a description of Slang – the *symbolic assembly language* for the RC 4000 computer. The present version of the assembler, called *Slang 3*, is designed to operate under the RC 4000 *Multiprogramming system* and the *File Processor*.

Chapter 1 contains a general description of the language. Chapter 2 introduces the basic constituents of Slang, which are: numbers, identifiers, expressions, instructions, textstrings, and directives. The overall structure of Slang programs is defined in Chapter 3, which explains the specification of bytes, words, and double words as well as the concepts of block and segment. The syntax of the language is defined by means of the Backus notation.

The manual is concluded by a description in Chapter 4 of the operation of the Slang 3 assembler.

The present language is not compatible with the one described in the first edition of the Slang manual (1966) due to the adoption of the ISO character set and certain changes in the operation codes and the assembly directives.

The Slang language was defined by the author; the implementation was done by *Torkild Glaven*.

CONTENTS

Chapter 1

GENERAL DESCRIPTION

1.1. Language Features

In Slang the contents of storage locations are described as instructions, numbers, or textstrings. *Numbers* can be of *real* or *integer* type. Instructions and data words can refer to other storage locations by means of symbolic addresses called *identifiers.* A symbolic address can be evaluated either *absolutely* or *relatively* with respect to the current load address.

An identifier can also refer to the value of an *expression* defined during assembly. Assembly expressions are written in conventional arithmetic notation using *parentheses* and the *operators* + - * / < (logical left shift) > (logical right shift) a. (logical and) and o. (logical or). The *operands* are integers or defined identifiers, i.e. identifiers to which values have been assigned either by previous expressions or through their use as labels.

With respect to the scope and usage of identifers, Slang has a *block structure* similar to Algol 60: identifiers must be declared in a block head before they are used. They can only be used within the block in which they are declared and in blocks enclosed by the former.

The outermost block, which contains all other blocks, delimits the entire *program.* Large programs can be divided into segments to facilitate assembly. A *segment* is a block for which binary output is produced immediately after the end statement.

To facilitate the handling of programs that exist in several versions, Slang includes *conditional assembly* in the following form: a piece of code can be headed by an expression, which is evaluated by the assembler. If the result is positive, assembly will continue after the expression. A negative result, however, will cause the assembler to skip the following statements up to a given delimiter.

1.2. Programming Examples

The following examples illustrate the overall structure of Slang programs. Let us first consider the following piece of code:

```
: procedure read char (char)
; comment: unpacks the next character from a storage
; address initialized by init read.
;          call:     return:
; w0                 char
; w1                 unchanged
; w2                 unchanged
; w3       link      link
```

```
b.i24                    ; begin
w.d18:  rx.w1   i1.      ;
        rx.w2   i2.      ;
        sh w1   0        ;  if readshift>0 then
        jl.     i0.      ;  begin
        al w1   -16      ;  readshift:=-16;
        al w2   x2+2     ;  readaddr:=readaddr+2;
  i0:   rl w0   x2+0     ;  end;
        ls w0   x1+0     ;  char:=word(readaddr) shift readshift;
        la.w0   i3.      ;  char:=char(17:23);
        al w1   x1+8     ;  readshift:=readshift+8;
c.(:c24>19a.1:)-1        ;  if testoutput then
        jd 1<11+28       ;  typew0(char);
z.      rx.w1   i1.      ;
        rx.w2   i2.      ;
        jl      x3+0     ;
  i1:   0, i2: 0         ; readshift: readaddr:
  i3:   8.177            ;
e.                       ; end
```

The texts preceded by semicolons are *comments* ignored by the assembler. The example shows a *block* of code enclosed by begin and end delimiters b. and e. The block begin is followed by a *declaration* i24 indicating that the programmer can use the names i0, i1, i2, - - - up to i24 as local identifiers within the block.

The following lines each describe a single machine instruction or data word. The delimiter w. means that they are assembled as *words* (in contrast to bytes and double words).

An *instruction,* like the following:

```
rx.w1  i1.
```

consists of a mnemonic *operation code* (rx meaning register exchange), a *modification part* (. meaning relative addressing), a *working register part* (w1 meaning working register 1), and an *address part* (i1. referring to the word labelled i1:). The period after the identifier i1 means that it is evaluated *relatively* during assembly.

Indexing is used, for example, in the instruction:

```
r1  w0  x2+0
```

i.e. register load in working register 0 the word addressed by index register 2.

Indirect addressing (not shown in the above example) is specified by enclosing the address part in parentheses:

```
jl.  (i5.)
```

i.e. jump with link relatively and indirectly via the word labelled i5:
Conditional assembly is illustrated by the statement enclosed in delimiters c. and z.

```
c.(:c24>19a.1:)-1
    jd  1<11+28
/.
```

The value of the *expression* (:c24>19a.1:)-1 determines whether the instruction jd 1<11+28 is assembled or skipped. It is assumed that the assembly condition has been defined previously in the program by *assignment* to the identifier c24:

```
c24=1<19  ; testoutput:=true;
```

or:

```
c24=0     ; testoutput:=false;
```

The next example illustrates the assembly of program data:

```
h. f2  : 8.3721, 115, c15      ; bytevalues:
w. f3  : <:load error:>        ; textstring:
f. f4  : -1.352'6              ; floating point number:
```

Bytevalues are prefixed by the delimiter h. The first value 8.3721 is an *octal number* with the digits 3721. It is followed by a *decimal number* 115. The last byte will get the value of the identifier c15, which must be defined elsewhere.
Textstrings are assembled in word mode and enclosed in brackets <: and :>, for example <:load error:>.
Floating-point numbers are prefixed by the delimiter f. The apostroph ' is used instead of the decimal radix 10 as shown by the example -1.352'6.
Instructions and data can be *labelled* by identifiers followed by a colon, for example f3:
Finally, we give the skeleton of a program consisting of two segments (enclosed by delimiters s. and e.) surrounded by a global block to illustrate the nesting of segments and blocks:

```
b. - - -                ; global block
   s. - - -             ; segment 1
      b. - - - e.       ; local blocks
      b. - - - e.       ;
   e.
   s. - - -             ; segment 2
      b. - - -          ; local blocks
         b. - - - e.    ;
      e.                ;
   e.                   ;
e.
```

Chapter 2

BASIC ELEMENTS OF SLANG

2.1. Character Set

2.1.1. Syntax.

<assembly character>::=<digit>!<letter>!<special character>!
<space>!<new line>

<digit>::= 0!1!2!3!4!5!6!7!8!9

<letter>::=
a!b!c!d!e!f!g!h!i!j!k!l!m!n!o!p!q!r!s!t!u!v!w!x!y!z!æ!ø!å!
A!B!C!D!E!F!G!H!I!J!K!L!M!N!O!P!Q!R!S!T!U!V!W!X!Y!Z!Æ!Ø!Å

<special character>::= "!£!$!%!&!'!(!)!*!+!,!-!.!/!:!;!<!=!>!?!_!`!{!
`!~!<apostrophe>

<space>::=<SP>

<new line>::=<NL>!<VT>!<FF>

2.1.2. Semantics.

Slang accepts a subset of the *ISO 7-bit character set* consisting of all *graphics* plus the layout characters *space*, *new line*, *vertical tabulation*, and *form feed*. All other ISO characters are treated as *blind* symbols by Slang.

2.1.3. Letters and Spaces.

Outside textstrings, Slang does not distinguish between *small letters* and *capital letters*, and all *spaces* are ignored.

2.1.4. Numerical Representation.

A table of the ISO character set and its numerical representation is given in Appendix A.

2.2. Numbers

2.2.1. Syntax.

<number>::=<real number>!<integer>!<radix number>

<real number>::=<decimal number>!<exponent>!<decimal number><exponent>

<decimal number>::=<integer>!<fraction>!<integer><fraction>

<fraction>::=.<unsigned integer>

<exponent>::='<integer>

<radix number>::=<radix><unsigned integer>

<radix>::=<unsigned integer>.

<integer>::=<unsigned integer>!+<unsigned integer>!-<unsigned integer>

<unsigned integer>::=<digit>!<unsigned integer><digit>

2.2.2. Semantics.
Slang accepts three types of numbers: real numbers, integers, and radix numbers. *Real numbers* and *integers* have their conventional meaning. The exponent part of a real number is expressed as an integral power of 10.
Radix numbers are expressed as a radix followed by a digitstring, for example, 2.110111 (a binary number) or 8.4357 (an octal number). The radix can be greater than 9, but in any case the number is converted digit by digit as follows:

number:=radix*number + digit

2.2.3. Examples.

Real:	33	+495'6	-0.67	'+8
Integer:	225	+17	-588	0
Radix:	2.101	5.3241	8.4692	129.58

2.3. Identifiers

2.3.1. Syntax.
<identifier>::=<declared identifier>!<load address>
<declared identifier>::=<identifier letter><unsigned integer>
<identifier letter>::= a! b! c! d! e! f! g! h! i! j
<load address>::= k

2.3.2. Semantics.
An *identifier* is a symbolic name given either to a *storage location* in the object program or to the value of an *expression* evaluated by the assembler.

2.3.3. Load Address.
The letter k is a reserved identifier for the *load address* of the current byte or word. K is initialized to zero before assembly.

2.3.4. Examples.

k	a0	d25	h123

2.4. Expressions

2.4.1. Syntax.

<expression>::=<sign><term>!<expression><operator><term>
<term>::=<operand>!(:<expression>:)
<operator>::=+!-!*!/!<!>!a.!o.
<sign>::=<empty>!+!-
<operand>::=<unsigned integer>!<radix number>!<identifier>!
<relative identifier>
<relative identifier>::=<declared identifier>.

2.4.2. Semantics.

An *expression* specifies the computation of an integer value during assembly. All *operands* and intermediate results are evaluated as 24-bit integers.

2.4.3. Identifiers.

An identifier used as a computational operand must have a value when the expression is evaluated, i.e. it must be defined either by a previous expression or through its use as a label (see Section 3.1.).
An identifier followed by a period is evaluated *relatively* to the current load address. For example the value of b2. is b2 - k

2.4.4. Operators.

The arithmetic *operators* + - * / have their conventional meanings of *addition, subtraction, multiplication,* and *division.* The *shift* operators < and > shift the left-hand operand logically left and right the number of bit positions specified by the right-hand operand. The Boolean operators a. and o. form the *and* and *or* combinations of the two operands bit by bit.

2.4.5. Precedence of Operators.

The sequence of operations in evaluating an expression is from left to right subject to the following rules of *precedence* among the operators:

first:	<>
second:	*/
third:	+-
fourth:	a.
fifth:	o.

This can be overruled, however, by the use of *parentheses* (: and :)

2.4.6. Examples.

```
-a5
k + 128
2.1011<20 + 6<18 + g4.
(:(:c35 – c0:)/5:)<3 o. 6.323
```

2.5. Instructions.

2.5.1. Syntax.

<instruction>::=<operation part><address part>
<operation part>::=
<operation code><relative mode><W register><indirect mode><X register>
<operation code>::= aa! ac! ad! al! am! as! aw! ba! bl! bs! bz! cf! ci! dl! ds! fa! fd!
fm! fs! hl! hs! ic! io! is! jd! je! jl! kl! ks! la! ld! lo! ls! lx! ml!
ms! nd! ns! pl! ps! rl! rs! rx! sc! sh! sl! sn! so! sp! ss! sx! sz!
wa! wd! wm! ws! xl! xs
<relative mode>::=<empty>! .
<W register>::=<empty>! w0! w1! w2! w3
<indirect mode>::=<empty>! (
<X register>::=<empty>! x0! x1! x2! x3
<address part>::=<empty>!<expression>

2.5.2. Operation Parts.

The *operation part* of an instruction is assembled as a 12-bit byte. It must begin with one of the *mnemonic codes* listed in Appendix B; *modifications* may follow in any order. If no modifications are specified, w0, x0, and direct addressing are understood.

2.5.3. Address Parts.

The *address part* of an instruction is evaluated by an *expression* and assembled as a 12-bit byte. It must be confined to the *range:*

$$-2048 <= \text{address part} <= 2047$$

An empty address part is loaded as a zero.

2.5.4. Indirect Addressing.

For aesthetic reasons, Slang permits the programmer to terminate an indirect address by a right parenthesis). This is treated as a blind symbol however.

2.5.5. Examples.

Operation Parts:

```
ac
rl w0
jd x3
bz. w3 (x2)
```

Instructions:

```
sz w1 2.1110
jl  e55
wa. w3. (x1+f15.)
ls w0 24-(:c1-c1/24*24:)
```

2.6. Textstrings

2.6.1. Syntax.
<text>::=<:<textstring>:>
<textstring>::=<text character>!<textstring><text character>
<text character>::=<assembly character>!<numerical character>
<numerical character>::=<<unsigned integer>>

2.6.2. Internal Representation.
The characters between the delimiters <: and :> are stored in consecutive words with 3 characters per word. Each character will be represented by its 7-bit numerical representation defined in Appendix A. Unused character positions at the end of the last word are filled with NULL characters.

2.6.3. Blind Characters.
All unprintable characters are ignored by Slang (Section 2.1.2). They can, however, be included in textstrings by means of the numerical notation described below.

2.6.4. Numerical Characters.
An unsigned integer enclosed in brackets < and > is interpreted as an 8-bit character with a numerical representation defined by the integer modulo 256.
This notation can be used to specify *control characters* in a textstring. For example:

null	<:<0>:>
acknowledge	<:<6>:>

Slang interpretes the character pair :> as a text terminator, and the character < followed by a digit as the beginning of a numerical character. The numerical notation can be used to include these symbols in the assembled textstring:

:>	<:<58><62>:>
<6>	<:<60>6>:>

2.6.5. Examples.

```
<:syntax error<0>:>
<:This text includes a
   New Line character:>
<:too big<10>try again :>
```

2.7. Directives and Comments

2.7.1. Syntax.

<directive>::=<input control>!<identifier listing>!<assembly jump>!
<dummy directive>!<testoutput control>!<end line>
<input control>::= t.! n.
<identifier listing>::= i.
<assembly jump>::= j.
<dummy directive>::= x.! y.
<testoutput control>::= o.! æ.
<end line>::=<new line>!;<line comment><new line>!
m.<line comment><new line>
<line comment>::=<any string not containing a new line>

2.7.2. Semantics.

Directives have no effect on the binary code produced but serve to control the assembly.

2.7.3. Input Control.

The delimiters t. and n. permit the switching between two alternative input mediums (originally called *typewriter* and *normal input*) during assembly. The exact functions of the input control depend on the surrounding system (see Section 4.5).

2.7.4. Identifier Listing.

The delimiter i. produces a listing of *identifier values* during assembly. The list includes only defined identifiers which are local to the current block.

2.7.5. Assembly Jump.

The delimiter j. causes a *jump* to the last assembled word during assembly. The result is unpredictable if the last word is undefined when the jump is made. This feature is intended for the implementation of system dependent functions outside the scope of the Slang language (see Section 4.7).

The assembly jump is only accepted within *word statements* (see Section 3.5).

2.7.6. Dummy Directives.

The delimiters x. and y. survive from a previous version of Slang. In the present version they can be used as *separators* within statements, but apart from that they have no directive effect on assembly.

2.7.7. Testoutput Control.

The delimiter o. causes the assembler to print internal testoutput during assembly; it can be turned off again by means of the delimiter æ. This facility is used to debug the assembler.

2.7.8. Line Comments.

Comments can be included in the program following the delimiter m. or a semicolon. This causes everything up to a new line to be ignored. Comments following the delimiter m. are *messages* which are displayed on the output medium during assembly.

Chapter 3

STRUCTURE OF SLANG PROGRAMS

3.1. Labels

3.1.1. Syntax.
<label>::=<declared identifier>:

3.1.2. Semantics.
Labels are used as symbolic addresses of *storage locations.* In *halfword* and *word statements* an identifier used as a label gets the current load address as its value (see Sections 3.4 and 3.5). In *floating-point statements* an identifier used as a label gets the current load address + 2 as its value (see Section 3.6).
An identifier can only be defined once as a label within the block in which it is declared. It can, however, be *redefined* by an assignment statement (see Section 3.2).

3.1.3. Examples.

```
a3:        e22:        j0:
```

3.2. Assignment Statements

3.2.1. Syntax.
<assignment>::=<identifier>=<expression>

3.2.2. Semantics.
An *assignment* statement assigns the value of an *expression* to an identifier. An identifier defined by an expression must not be *redefined* as a label within the block in which it is declared. It can, however, be redefined by another assignment.

3.2.3. Load Address Control.
The *load address* k can be changed explicitly by an assignment statement. This effects the values of subsequent identifiers used as labels.
The binary code produced does not contain explicit information about the change of load address to be used when the program is actually loaded and executed. The conventions for program loading in a given system is outside the scope of the Slang language.

3.2.4. Examples.

```
k = 8
k = k - 130
c6 = b2. - (:g32 + 4:)
i4 = 2.101<21 + 5
```

3.3. Compound Statements

3.3.1. Syntax.

<compound statement>::=
h.<prelude>!h.<prelude><halfword compound><epilog>!
w.<prelude>!w.<prelude><word compound><epilog>!
f.<prelude>!f.<prelude><floating-point compound><epilog>
<prelude>::=<empty>!<label>!<directive>!<prelude><prelude>
<epilog>::=<empty>!<separator>
<separator>::=,!<directive>!<separator><prelude>

3.3.2. Semantics.

A Slang program consists of a sequence of simple statements specifying numbers, instructions, textstrings, and assignments. To distinguish between the assembly of *halfwords*, *words*, and *double words*, Slang employs three types of *compound statements*. These are prefixed by the delimiters h. w. and f. The full extent of compound statements will become clear in the following.

3.3.3. Separators.

Slang uses a *comma* to separate one statement from another. A separating comma can be replaced or followed by a sequence of *directives*, *comments*, or *new lines*. *Labels* can also be inserted after a separator.

3.3.4. Prelude and Epilog.

The first statement within a compound statement can be prefixed by any sequence of *labels* and *directives*. The last statement can be terminated by a *separator* if desired.

3.3.5. Address Rounding.

A delimiter h. that is followed by a prelude only has no effect on the program. The delimiters w. and f. have the effect of setting the load address k equal to the nearest word address. The statements have no effect if k is even; if k is odd, a byte value zero is loaded and k is increased by 1.

3.3.6. Examples.

Prelude:	a5:	i.	b2:g3:x.c27:n.	m. message comment
Separator:	,	t.	,g15:y.n.a9:h0:	' line comment

3.4. Halfword Statements

3.4.1. Syntax.

<halfword compound>::=<byte>!<assignment>!r.<expression>!
<halfword compound><separator><halfword compound>
<byte>::=<operation part>!<expression>

3.4.2. Semantics.

The *operation parts* and *expression values* that follow the delimiter h. are loaded into consecutive halfwords. After each byte, the *load address* k is increased by 1.
A byte value must be confined to the range:

-2048 <= byte <= 4095

3.4.3. Undefined Bytes.

A byte can be an identifier whose value is defined later in the program either by an assignment or through its use as a label. An *undefined byte* can only have the following format:

<undefined byte>::=<declared identifier>!+<declared identifier>!
<relative identifier>!+<relative identifier>

3.4.4. Assignments.

Assignments to the *load address* k are not permitted within halfword statements.

3.4.5. Byte Repetition.

An expression following the delimiter r. is evaluated by the assembler. If the result is greater than zero the last assembled byte is also loaded into the following <expression>-1 bytes. The byte value is unpredictable if the last byte contains an undefined identifier.

3.4.6. Examples.

```
    h. b3: al w2,    rs w3 x2,    se. w0 (x3)      ; this is a
                                                   ; comment
          294,          -45,    a5:d8: 2.1011
         -c66,         f99.t.   (:j2 - j0:) /5
```

3.5. Word Statements.

3.5.1. Syntax.

<word compound>::=<word>!<text>!<assignment>!r.<expression>!
<word compound><separator><word compound>
<word>::=<instruction>!<expression>

3.5.2. Semantics.

The *instructions* and *expression values* that follow the delimiter w. are loaded into

consecutive storage words. After each word, the *load address* k is increased by 2. A word value must be confined to the *range:*

-8 388 608 <= word <= 8 388 607

3.5.3. Undefined Words.
A data word or an address part can be an undefined identifier in the same sense as defined in Section 3.4.3.

3.5.4. Instructions with Relative Addressing.
In an instruction, the period for *relative addressing* is set independently in the operation part and the address part. For example:

```
w. a0: jl. w2 b0.
```

It is important to note that the instruction word above is not equivalent to the following halfword statement:

```
h. a0: jl. w2. b0.
```

In the first case, the value of the relative identifier b0. is: b0 - k = b0 - a0, whereas in the second b0. becomes: b0 - k = b0 - (a0 + 1)

3.5.5. Word Repetition.
An expression following the delimiter r. is evaluated by the assembler. If the result is greater than zero the last assembled word is also loaded in the following <expression>-1 words. The word value is unpredictable if the last word contains an undefined identifier.

3.5.6. Examples.

```
w. t. g5: 2.101, -8 388 608, j87 a. (:a4 o. 6:)
        <:ready<10><0>:>, a2:a3: h9=g35<2-11 i. 25, r.8
    ; the following line shows examples of instructions:
        jl x1 8, b4: rs. w2 (d4.), al w0 x2 +16
```

3.6. Floating-point Statements

3.6.1. Syntax.

<floating-point compound>::= <real number>! <assignment>! r.<expression>!
<floating-point compound><separator><floating-point compound>

3.6.2. Semantics.
The *real numbers* that follow the delimiter f. are loaded into consecutive double words as normalized floating-point numbers. After each real the *load address* is increased by 4.

3.6.3. Real Repetition.
An expression following the delimiter r. is evaluated by the assembler. If the result is greater than zero the last assembled double word is also loaded in the following <expression>-1 double words.

3.6.4. Examples.

f. k=2000, c2: 233, 495'6, c3:e7: −0.45 t. +9.34' −49, r. a6*2

3.7. Programs, Segments, and Blocks

3.7.1. Syntax.

<program>::=<program comment><block>
<program comment>::=<any string not containing s. or b.>
<block>::=<block head><block tail>
<block head>::=
s.<prelude>!s.<prelude><head compound><epilog>!
b.<prelude>!b.<prelude><head compound><epilog>
<head compound>::=<declaration>!<assignment>!
<head compound><separator><head compound>
<declaration>::=<declared identifier>
<block tail>::=e.!<compound statement><block tail>!
<block><block tail>

3.7.2. Semantics.
A *block* is a sequence of compound statements enclosed by one of the delimiter pairs s. and e. or b. and e. Blocks can be nested. The *block* concept is introduced to control the *scope of identifiers* in the following way: first, all identifiers must be declared in a block head before being used; second, identifiers can only be used within the block in which they are declared and in blocks enclosed by the former; third, if the same identifier is declared in several nested blocks it will function as several identifiers, each of which can only be used within its own block.

3.7.3. Declarations.
At each *block entry,* the assembler reserves a table to contain the values of identifiers declared in that block. A *declaration* consists of one of the letters a through j followed by an unsigned integer. This permits the programmer to use any identifier beginning with the selected letter followed by an integer less than or equal to the limit specified. For example, the declaration:

b. c27, a9

initializes a block in which the identifiers c0 to c27 and a0 to a9 can be used. A *block end* e. deletes all local identifier values from the internal table.

3.7.4. Programs and Segmentation.
The outermost block delimits the *program* to be assembled. A program can be divided into segments. A *segment* is a block headed by the delimiter s. At the end of a segment, Slang outputs the entire segment as binary machine code. (It should be noted that binary output is only produced for segments). Segmentation imposes the following restriction on the use of identifiers: identifiers declared in blocks surrounding a segment must be defined before they are used within the segment.

3.7.5. Program Comment.
The input control directives t. and n. as well as line comments preceded by a semicolon are allowed before the first block head of the program; all other characters are, however, treated as a comment to the program.

3.8. Conditional Assembly

3.8.1. Syntax.
<condition head>::= c.<expression><separator>
<condition end>::= z.

3.8.2. Semantics.
Conditional assembly of a piece of Slang code is specified by the delimiter c. followed by an *expression* that is evaluated by the assembler. If the result is greater than or equal to zero, assembly continues after the expression. A negative result indicates that the following statements up to the delimiter z. are to be skipped.
Assembly conditions can be *nested* but do not necessarily follow the block structure of the program, i.e. a sequence such as:

c. - - b. - - z. - - e.

is accepted.
The *condition head* and *condition end* can be used in all places where *directives* are accepted.

Chapter 4

OPERATION OF THE ASSEMBLER

4.1. Introduction

This chapter describes the features and the operation of the *Slang 3 assembler* designed for the RC 4000 multiprogramming system. The assembler is called and controlled by the *file processor* (fp). In the following sections it is assumed that the reader is familiar with the fp manual.

The assembler takes its input from one or more *text files* described in the catalog on the backing store or in the fp notes; it outputs a *binary program* consisting of one or more Slang segments on a single file.

The input media can be any media handled by fp, whereas the output medium must be either backing store or magnetic tape.

4.2. Size and Speed

Slang and the fp occupy about 4000 words of the internal store (including buffers for the current input and output media). The rest of the available store is divided between the assembled program and the table of identifiers as follows:

Assembled Program:

code = run time size
k assignment = 2 words

Identifier Table:

block head = 3 words
declaration = 1 + 2*identifier index words

The internal assembly speed is about 5000 characters per second.

4.3. Calling the Assembler

FP will load and start the assembler when it reads a *call* of the following format from the current input medium:

<result> slang <parameter list>

The *result* can be empty or of the form:

<fp note> =

or <files> =

The *parameter list* can be empty or contain one or more parameters of the form:

<source file>

or <assembly option>.yes

or <assembly option>.no

The assembler can input the *source program* from one or more text files given in the parameter list: when the first source file is exhausted, input is taken from the next source file, and so on. A source file is identified by a name referring to either a catalog

entry or an fp note, which describes the actual input document and its initial position. If no source file is specified, input is taken from the current input medium; if this medium is exhausted, assembly is terminated.

The assembler can output the *object program* on a single result file. If *no result* file is specified, the assembler does not output binary code. If the result file is identified by the name of an *fp note* and the note is empty (i.e. does not describe a document), a temporary area of a standard size is created on the backing store and described in the note. After assembly the size of the area is decreased in accordance with the actual size of the object program. If the note already describes a document or if the result file is identified by the name of a catalog entry, which describes a document, binary code is output on that document.

4.4. Format of Binary Program

The format of an assembled object program depends on the kind of the output document:

On *backing store* the binary segments are output word by word from the beginning of the area, ignoring the boundaries of backing store segments.

On *magnetic tape* each binary segment is output as one block; the last block is terminated by a tape mark.

The present assembler can only deliver output on backing store or magnetic tape. A separate utility program, *binout,* must be used to copy binary output on *paper tape* with the following format: Each word is output as four 6-bit characters with odd parity; each binary segment is terminated by a 7-bit character with odd parity in which the left-most bit is one while the right-most six bits form a checksum of all other characters in the segment modulo 64.

4.5. Assembly Options

The following is a list of assembly modes that can be turned on or off by explicit parameters to Slang:

type.yes type.no	Defines whether the delimiter t. should cause the assembler to take *input* from the current input medium. The delimiter n. returns the assembler to the normal source file given in the parameter list.
list.yes list.no	Defines whether the assembler should print the *source text* (including the "typed" input) on the current output medium.
names.yes names.no	Defines whether the delimiter i. should cause the assembler to print *identifier values* on the current output medium.
message.yes message.no	Defines whether the delimiter m. should cause the assembler to print a *message comment* on the current output medium.

warning.yes
warning.no — Defines whether the assembler should print *warning messages* on the current output medium.

entry.yes
entry.no — Defines whether the assembler should change the *result descriptor* if no errors are found during assembly. The descriptor is changed as follows:

```
content:= 2;
entry:= 4;
length:= total number of bytes output;
```

remove.yes
remove.no — Defines whether the assembler should *remove* the catalog entry for a *temporary backing store area* used as result file if errors are found during assembly.

If no assembly options are defined in the parameter list, the following *initial values* are used:

```
type.no
list.no
names.no
message.yes
warning.yes
entry.yes
remove.yes
```

4.6. Assembly Messages

During assembly, certain control and error messages will be printed on the current output medium. Most of these include the value of the load address k. This in combination with a listing of identifier values facilitates the identification of trouble spots.

The following are normal *control messages:*

<k> type
: The delimiter t. causes switching from a source file to the current input medium.

<message comment>
: The delimiter m. causes printing of a message comment.

;<k> id list
b. <declarations>
<id list>
: The delimiter i. causes printing of declarations and identifier values local to the current block or segment.

slang ok <slang segments>/<bytes>/<backing store segments>
Slang returns to the fp and sets the ok bit to true. The size of the assembled program is expressed as the number of Slang segments, bytes, and backing store segments output.

During a *call* of the assembler the following messages can appear:

***slang param <illegal parameter>
Illegal parameter syntax. The parameter is ignored.

***slang work area connect <result>
The result file cannot be connected for output. Assembly is performed without binary output.

***slang work area kind <result>
The kind of the result file is neither backing store nor magnetic tape. Assembly is performed without binary output.

During assembly *warnings* are printed in the following cases:

<k> cancel
Input of a CAN character from typewriter causes all previous characters on the current text line to be skipped.

<k> illegal <character value>
Input of a blind control character other than NUL, CR, or DEL.

<k> file mark
Troubles encountered during output of final tape mark on magnetic tape.

<k> byte value
A byte value is outside the range −2048 to 4095; or an address part has a value outside the range −2048 to 2047. (The address part of a jd instruction can, however, be in the range −2048 to 4095. Thus monitor calls will not cause warnings during assembly.)

<k> relative
A relative identifier is used as address part in an instruction without relative addressing.

<k> repetition
A repetition expression with a value less than 1.

The following *error messages* cause everything up to the next separator to be skipped:

<k> syntax

Illegal structure of delimiters and operands.

<k> <identifier> declaration

An identifier is declared twice in the same block head, or a declaration index exceeds 4095.

<k> <identifier> undeclared

An identifier is used without being declared.

<k> <identifier> definition

An identifier that has been defined elsewhere by a label or an assignment is redefined by a label within the same block.

<k> <identifier> undefined

An identifier is undefined when an expression is evaluated; or an identifier declared outside of a segment is undefined when it is used within the segment.

<k> undefined at end

One or more identifiers that have been used as bytes or words are undefined on exit from the block in which they are declared. Each undefined identifier is followed by the addresses at which it has been used.

<k> program too big

The binary output exceeds the capacity of the result file. Assembly continues without binary output.

The following situations cause immediate *termination of assembly:*

<k> stack

The size of the object code and the identifier table exceeds the available internal store. This can be remedied either by using a larger internal store during assembly or by dividing the source program into segments.

<k> connect <source file>

A source file cannot be connected for input.

<k> no text <source file>
A source file contains a character with a value > 127.

<k> end source
The last source file is empty before the logical program end.
The missing number of z. and e. are generated by the assembler.

<k> jump
Working register 2 <> 0 on return from a jump action (see Section 4.7).

<k> slang fault
Caused by a programming error in the assembler. Please send a listing of the source text and the assembly messages to Regnecentralen.

If the assembler has printed one or more error messages or a termination message during assembly, it will return to the fp at the end of the program and set the ok bit to false. In this case the *final assembly message* is:

***slang sorry <slang segments>/<bytes>/< backing store segments>

If the source text is empty before a block begin (b. or s.) is read, the ok bit is also set to false and the assembly message is:

***slang no program

4.7. Jump Action

When the assembler reads the delimiter j. it performs a jump to the last assembled word with the following register values:

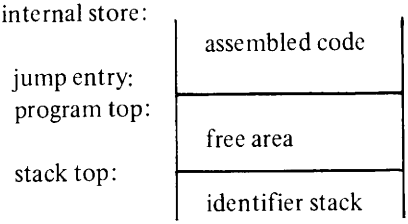
w1: fp base
w2: slang base
w3: return address

All entries in the fp are defined relative to the *fp base* as described in the manual of the file processor.

The *slang base* is the address of a table within the assembler, which contains the following entities:

slang base:	program top (first free word)
+ 2	stack top (last free word)
+ 4	result note address (−1 if no note)
+ 6	last k assignment
+ 8	result name address (−1 if no name)

The meaning of the addresses *program top* and *stack top* is illustrated by the following map of the internal store:



The *result note address* points to the fp note describing the object program file. The format of the note is defined in the manual of the file processor.

The *last k assignment* is the address of a word defining the value of the last k assignment. The current value of k (corresponding to the next word to be assembled in program top) can be calculated as follows:

```
rl    w1    x2+0    ; k:=   program top
ws    w1    x2+6    ;       – last k assignment
al    w1    x1–2    ;       – 2
wa    w1    (x2+6)  ;       + word (last k assignment);
```

The *result name address* is the address of the first of four words defining the name of the catalog entry, which describes the object program file.

On *return* from a jump action the assembler interprets the registers as follows:

If w0 <> 0 the textstring addressed by the register is printed up to a NUL character on the current output medium.

If w2 <> 0 assembly is terminated by the jump message (see Section 4.6); otherwise assembly continues.

The following example shows a jump action that looks up and changes the catalog entry describing the result file. At the end the jump action decreases the program top with the size of the jump code, thus preventing it from being included in the final program:

```
b. a4                                   ; begin
w. a0:  rs.   w3   a3.                  ; jump action:
        al.   w1   a2.                  ;
        rl    w3   x2+8                 ;
        jd    1<11+42                   ;  look up entry (result name, tail, result),
        se    w0   0                    ;  if result <> 0
        jl.        a1.                  ;  then terminate slang;
        -----                           ;  change tail as desired;
        jd    1<11+44                   ;  change entry (result name, tail, result);
        se    w0   0                    ;  if result <> 0
        jl.        a1.                  ;  then terminate slang;
        rl    w1   x2+0                 ;
        al    w1   x1+a4                ;  program top:=
        rs    w1   x2+0                 ;  program top - size of jump action;
        al    w2   0                    ;
  a1:   al    w0   0                    ;  continue slang;
        jl.        (a3.)                ;
  a2:   0,    r.10                      ; tail:
  a3:   0                               ; return address:
        jl.              a0.            ; entry point:
  a4=a0. ,k=a0                          ; size of jump action:
        j.                              ;  goto jump action;
e.                                      ; end
```

APPENDIX A: THE ISO 7 BIT CHARACTER SET

The character set contains 128 7-bit characters. In the numerical representation of any one character the bits are identified by:

b7 b6 b5 b4 b3 b2 b1

which have the following significance in the binary system:

64 32 16 8 4 2 1

In the code table below the columns and rows are identified by the decimal equivalent of the following binary numbers:

column: b7 b6 b5 0 0 0 0

row: 0 0 0 b4 b3 b2 b1

Accordingly, the decimal value of a character is the sum of the column and row numbers. For instance, the character H has the numerical representation 64 + 8 = 72. Empty positions in the code table specify characters that are ignored by Slang.

	0	16	32	48	64	80	96	112
0			SP	0	@	P	`	p
1			!	1	A	Q	a	q
2			"	2	B	R	b	r
3			£	3	C	S	c	s
4			$	4	D	T	d	t
5			%	5	E	U	e	u
6			&	6	F	V	f	v
7			'	7	G	W	g	w
8			(	8	H	X	h	x
9		EM	)	9	I	Y	i	y
10	NL		*	:	J	Z	j	z
11	VT		+	;	K	Æ	k	æ
12	FF		,	<	L	Ø	l	ø
13			-	=	M	Å	m	å
14			.	>	N	^	n	¯
15			/	?	O	_	o	

APPENDIX B: MNEMONIC OPERATION CODES

The following is a list of mnemonic codes for the instruction set of the RC 4000. The two columns give the mnemonic code and the complete name of each instruction.

AA	Add Integer Double Word
AC	Load Address Complemented
AD	Shift Double Arithmetically
AL	Load Address
AM	Modify Next Address
AS	Shift Single Arithmetically
AW	Autoload Word
BA	Add Integer Byte
BL	Load Integer Byte
BS	Subtract Integer Byte
BZ	Load Byte with Zeroes
CF	Convert Floating to Integer
CI	Convert Integer to Floating
DL	Load Double Register
DS	Store Double Register
FA	Add Floating
FD	Divide Floating
FM	Multiply Floating
FS	Subtract Floating
HL	Load Half Register
HS	Store Half Register
IC	Clear Interrupt Bits
IO	Input Output
IS	Store Interrupt Register
JD	Jump with Interrupt Disabled
JE	Jump with Interrupt Enabled
JL	Jump with Register Link
KL	Load Protection Key
KS	Store Protection Key
LA	Logical And
LD	Shift Double Logically
LO	Logical Or
LS	Shift Single Logically
LX	Logical Exclusive Or
ML	Load Mask Register
MS	Store Mask Register
ND	Normalize Double
NS	Normalize Single
PL	Load Protection Register
PS	Store Protection Register
RL	Load Register
RS	Store Register
RX	Exchange Register and Store
SE	Skip if Register Equal
SH	Skip if Register High
SL	Skip if Register Low
SN	Skip if Register Not Equal
SO	Skip if Register Bits One
SP	Skip if No Protection
SS	Subtract Integer Double Word
SX	Skip if No Exceptions
SZ	Skip if Register Bits Zero
WA	Add Integer Word
WD	Divide Integer Word
WM	Multiply Integer Word
WS	Subtract Integer Word
XL	Load Exception Register
XS	Store Exception Register

INDEX

Printed in Denmark by
A/S F. Andersen & Son, Computer Dept.
Lay-out Freddi Schlechter

A/S REGNECENTRALEN

SCANDINAVIAN INFORMATION PROCESSING SYSTEMS

HEADQUARTERS: FALKONER ALLÉ 1 · DK-2000 COPENHAGEN F · DENMARK
PHONE: (01) 10 53 66 · TELEX: 6282 RCHQ DK · CABLES: REGNECENTRALEN

AUSTRIA
BENELUX
DENMARK
GERMANY
NORWAY
SWEDEN