AFL Agent Forth

Table of Contents

Introduction

Programming

Content

AFL Overview

Name

AFL Programming Language - Agent Forth (αFORTH) Overview

Synopsis

```
par p TYP
var x TYP
:A1 .. ;
:A2 .. ;
:f .. ;
:$S .. ;
:%TRANS
 |A1 ε ?A2 ε ?A3 ..
 |A2 ..
;
```

Description

Long description.

Stacks

```
Data Stack
 ( v1 v2 v3 -- r1 r2 r3 )

Return Stack
R( v1 v2 v3 -- r1 r2 r3 )
```

The top of the stack is the right underlined element of the group (i.e., r3). It is assumed that the data width of the data and return stack is equal (n bit) and that the data width is equal to the instruction code size to enable code morphing support by using the data stack. Common data- and instruction code widths are 16 and 24 bits.

Code Frame

The code frame is a self-contained unit which holds all code and persistent data. A code frame consists of a boot section, which mainly reflects the control state of the program and which can be modified by using code morphing operations. The code frame provides self initialisation by executing instructions in the boot section, by executing instructions in the code frame body, and by definition instructions (word, variable, transition) within the code frame. A code frame will always start execution at the top of the frame by executing the boot section. A newly created or migrated code frame will pass through the entire code frame until the transition section is reached. The transition section has a boot header, too, which can be modified, and branches to the next activity row to be executed.

```
----------------------------------------
1. Boot Section
 B1 B2 B3 B4 B5 B6 B7 B8 .. B16
----------------------------------------
2. Lookup and Relocation Table LUT [N]
 FLAG OFF FRAME SEC
 FLAG OFF FRAME SEC ...
----------------------------------------
3. Variable Definition
 VAR Vi TYP [N]
4. Initialization Instructions
 C1 C2 C3 ..
5. Activity Word Definition
 :*Ai REF REF .. ;
6. Function Definition
 :Fi .. ;
7. Signal Handler Definition
 :$Hi .. ;
----------------------------------------
8. Transition Table Definition Sections
 :%Ti B1 B2 B3 B4
   |Ai {1 .. ?Ai+1} {2 ..} ..
   |Ai+1 {1 .. ?Ai+2} {2 ..} ..
   ..
 ;
----------------------------------------
```

Code frame format

Program Structure

AFL	AML	Description

par p1 int par p2 int .. var x int var y int ..	VAR VAL(LUT#) VAL(size) DATA .. VAR VAL(LUT#) VAL(size) DATA ..	Definition of agent parameters and variables
:*act .. ;	DEF VAL (LUT#) VAL (size) .. EXIT	Definition of an activity word
:func .. ;	DEF VAL (LUT#) VAL (size) .. EXIT	Definition of a generic function word
:$handler .. ;	DEF VAL (LUT#) VAL (size) .. EXIT	Definition of a signal handler word
:%trans |act1 .. ?act2 .. . |act2 .. ?act3 .. . ; trans	TRANS VAL (LUT#) VAL (size) NOP NOP NOP NOP TCALL 11 ..	Definition of a transition table word and default transition table call.

AFL Program Structure

Code Lookup Table

The program code embeds a lookup table (LUT) with relocates code addresses of variables, activity, and function words. The LUT consists of rows, each consisting of four columns: { FLAG, OFF, FRAME, SEC }.
The FLAG column specifies the kind of the LUT row entry { FREE, VAR, ACT, FUN , FUNG, TRANS}. The OFF and FRAME columns specify code addresses, whereas the SEC column entry is used for auxiliary values, mainly for caching of transition section branches related with activity words. The SEC column can be packed with the FLAG field optimising resource requirements.
The first row of the LUT always contains the relocation data for the current

transition table word used to load the TP register which points to the start of the boot section of the transition table.

Virtual Machine Instruction Set

Only a sub-set AML of all available AFL operators are implemented on VM instruction set level. They are added with a AML column in the following instruction set tables.
The instruction code format is divided into a short and a long code format. The short code enables code packaging in one instruction word to speed up code processing. The short code format is 8 bit wide, the long code format is equal to the full code and data word width (e.g., 16 or 24 bits). The first two highest bits determine the code format. The long code format is used by instructions with arguments, like for example, branch or value literal words, indicated in the pseudo notation by enclosing the argument in parentheses, i.e., BRANCH(-100).

KIND	SELECTOR	OPCODE	ARGUMENT	VALUE
Value	1X	-	-	(N-1) data bits
Short Command	01	6 bits	-	-
Long Command A	00	01 10 11	(N-4) data bits	-
Long Command B	00	00XXX	(N-7) data bits	-

Instruction code format (N: instruction word and data width, LSB format)

Mathematical Operations and Values

The set of mathematical operators consists of arithmetic, relational, and logical operations. The operands are retrieved from the data stack and the result is stored on the data stack again.

AFL Value	AML	Stack	Description
n 0xXXX 0bBBB 0oOOO	VAL(n) $80+n	(-- v)	Pushes a signed constant value to the data stack. The value v can be in the range {-2n-2..2n-2-1} with n bit data width of the code instruction.

N 0xXXX 0bBBB 0oOOO	VAL(n1) $80+n1 EXT(sign) $06+sign	(-- v)	Pushes an unsigned constant value to the data stack followed by a MSB sign extension (sign=+-1) word for large values. The value v can be in the range {-2n-1..-2n-2-1} or {2n-2..2n-1-1} with n bit data width.
0/1	ZERO/ONE $40/$41	(-- v)	Pushes value zero or one to the data stack.
long	EXT(LONG) $06+$10	(--)	Long operation prefix: values can be combined to double word-size values, and artihm., log., and rel. operations can be extended to use the long word values.

Value literals

AFL Operator	AML	Stack	Description
+	ADD $42	(v1 v2 -- r)	Addition (signed) r = v1+v2
-	SUB $43	(v1 v2 -- r)	Subtraction (signed) r = v1-v2
*	MUL $44	(v1 v2 -- r)	Multiplication (signed) r = v1*v2
/	DIV $45	(v1 v2 -- r)	Division (signed) r = v1/v2
mod	MOD $46	(v1 v2 -- r)	Division (signed) returning remainder r = v1 % v2
negate	NEG $47	(v -- r)	Negate operand r=-v
abs	-	(v -- r)	Return positive equivalent r= if v1<0 then -v1 else v1
min	-	(v1 v2 -- r)	Return smallest number r= if v1<v2 then v1 else v2

max	-	(v1 v2 -- r)	Return biggest number r= if v1>v2 then v1 else v2
random	RANDOM $69	(min max -- r)	Return a random number in the range interval [*min*,*max*]

Arithmetic Operations

AFL Operator	AML	Stack	Description
<	LT $4A	(v1 v2 -- r)	Lower than (signed) r = (v1<v2)
>	GT $4B	(v1 v2 -- r)	Greater than (signed) r = (v1>v2)
=	EQ $4C	(v1 v2 -- r)	Equal (signed) r = (v1=v2)
<>	-	(v1 v2 -- r)	Not equal (signed) r = (v1<>v2)
<=	LE $4D	(v1 v2 -- r)	Lower than or equal r = (v1<=v2)
>=	GE $4E	(v1 v2 -- r)	Greater than or equal r = (v1>=v2)
0=	-	(v -- r)	Test for zero
0<>	-	(v -- r)	Test for non zero
within	-	(v1 v2 v3 -- r)	Test if v2 is within [v1,v3]

Relational Operations

AFL Operator	AML	Stack	Description
and	AND $50	(v1 v2 -- r)	r = v1 $\wedge$ v2
or	OR $51	(v1 v2 -- r)	r = v1 $\vee$ v2
xor	-	(v1 v2 -- r)	r = v1 $\oplus$ v2
not	NOT $52	(v -- r)	r = $\neg$v
invert	INV $53	(v -- r)	r = $\forall$bits b_i(v): $\neg b_i$
lshift	LSL $56	(v n -- r)	r = shift v left by n bits
rshift	LSR $57	(v n -- r)	r = shift v right by n bits

Logic bitwise Operations

The following AFL data types and value formats are supported:

```
type DT = {
  bool, char, int(eger), float, int32, word, string
}

Decimal Format, integer:        0,1,2..
Binary Format, integer:         0b101
Hexadecimal Format, integer:    0x13F
Hexadec. Format Constructor:    0x{E1,E2,3,...} => 0x123...
                                const E1 1 ...
                                enum XYZ E0 E1 E2 ...
Octal Format, integer:          0o175
Octal Format Constructor:       0o{E1,E2,3,...} => 0o123...
                                const E2 2 ...
                                enum XYZ E0 E1 E2 ...
Floating Point Format, float:   1.234
Exponential Format, float:      1.2E-10
Char Format, char:              'c'
String Format, string:          "abcd"
Boolean Format, bool:           true, false
```

The format constructors are used to create packed values composed of symbolic constants or numbers.

Enumeration Types

An enumeration type definition can be used to define a set of numbered symbolic constants. An optional start index value for the first symbol can be given ([start]). The symbolic constant are replaced in expressions by their respective number value.

```
enum ENUMTYPE SYM1 SYM2 .. ;
enum STARTINDEX ENUMTYPE SYM1 SYM2 .. ;
```

Enumerations are used to map symbolic names on integer constant values. The first symbol element of an enumeration is either assigned to an index value zero or the value of an optional start index *STARTINDEX*. For each following symbol the index value is incremented by one.

Control Flow

Control structures are used to control the program flow either by down-directed branching or by using up-directed loops.

AFL Control Structure	AML	Stack	Description

`if`
`true words`
`then`
	-	`( flag -- )`	If the flag value on top of the stack is non-zero (true), the words between `if` and `then` are executed.

`if`
`true words`
`else`
`false words`
`then`
	-	`( flag -- )`	If the flag value on top of the stack is non-zero (true), the words between `if` and `else` are executed, otherwise the words between `else` and `then`.

`case key`
` v1 of .. endof`
` v2 of .. endof`
` ..`
` default words`
`endcase`
	-	-	Value-based case-select statement. The value of the key is compared with a list of possible values {*v1*, *v2*, ..}. If a value matches the key, the words between `of` and `endof` are executed.

`exit`	`EXIT`
	`$60`
		`R( ip cf`
		`-- )`
		` ( -- r1`
		`r2 .. )`
			The `exit` word causes a return from the current definition call. The code branch point and code frame are taken from the return stack. An exit call with an empty return stack kills the process.

-	`NOP` `$61`	`( -- )`	No operation instruction

-	BRANCH(Δ) $10+Δ	(--)	Branch code execution relative to current code position *IP* in the same code frame
-	BRANCHZ(Δ) $20+Δ	(flag --)	Branch code execution relative to current code position IP in the same code frame iff the flag on the data stack is zero (false).
-	CALL(#) $0E+#	R(-- ip cf)	Call a word and push the current code position IP+1 on the return stack and the current code frame, which is required by exit. The word code address and the code frame are taken from the LUT[#]. A call of a transition word sets the current active transition table and copies LUT entry to LUT row 0! No call frame is pushed!
-	BRANCHL $5C	(ip cf --)	Performs a long branch to a different code frame. Negative code frame numbers are relative to the root frame (*cf*=-1: root frame).

{*n .. } (enabled) {n .. } (disabled)	BBRANCH(Δ) $00+Δ	(--)	Dynamic blocks: a conditional branch which can be activated (Δ>0, block disabled) or deactivated (Δ<0, block enabled). If disabled, the block spawned by Δ is skipped. E=0 and Δ=0 is equal to the DATA word behaviour (NOP data marker and place holder)!

Branch control structures

AFL Control Structure	AML	Stack	Description
do .. loop do .. loop+	-	(limit index0 --) (limit index0 --) (increment) R(-- i)	The words between do and loop are repeated as long as *index* < *limit*. Loop increments a copy of the index counter on the top of the return stack.
begin .. again	-	(--)	This unconditional loop finishes only if an exception is thrown.
begin .. until	-	(--) (flag --)	This loop is executed at least one time, and repeats execution as long as the flag condition is false.
begin .. while .. repeat	-	(--) (flag --)	The loop starts at `begin` and all the words up to `while` are executed. If the flag consumed by while is true, the words between while and repeat are executed, finally looping again to begin.

`i j k`	`-`	`( -- index )`	Pushes the current loop index value on the data stack. The innermost first-level loop is referenced with *i*, and outer higher level loops with *j* and *k*.

Loop control structures

Code and Call Frames

It must be distinguished:

Code Frame Offset CFO
This is the absolute code address offset of a program code frame in the current code segment *CS*. The code segment CS is partitioned into fixed size code frames.

Code Frame Number CFN
This is the index number of the code frame. *CFN=CFO/CF_SIZE*.

Negative Code Frame Number CFN
Negative code frame numbers are relative to the root code frame of the current process. The root frame has number -1, the next linked frame has number -2, and so on. Code frames are linked if the last word of a code frame is a *NEXT*(*CFN*) instruction!

Each time a word is called, a call frame is stored on the return stack. This call frame consists of a tuple (*ip*,*cf*), which points to the return address of the next word to be executed after the call. There are two different word calls: transition calls using `TCALL` and generic function word calls using the CALL instruction.
In the case of activity word calls from within the transition table section using `TCALL` the current original (absolute) code frame offset taken form the *CF* register must be converted to a relative code frame number. On return, the call frame must be converted again to an absolute code offset to load the *CF* register again. This relative code frame numbering is required for code and process migration support. After migration absolute code frame offsets and numbers will change and may never be part of the data state of the process before migration, that means, stored on the stacks! Relative code frames other than the root frame are expensive to process because the code frame list must be iterated each time.

Code Frame Control and Modification

AFL	AML	Stack	Description
c>	FROMC W1 W2 ... $72	(n -- c)	Push the n following code words on the data stack
v>c	VTOC $5D	(v n off -- off') CS[cfs+off]: cv .. CS[cfs+off+1]: EXT ..	Convert n values from the data stack in a literal code word and extension if required. The new code offset after the last inserted word is returned.
>c	TOC $73	(c1 c2 .. n off --) CS[cfs+off]: c1 c2 .. cn	Pop n code words from the data stack and store them in the morphing code frame starting at offset off.
s>c	STOC $54	(.. off -- off') R(.. --)	Convert all data and return stack values to code values in the morphing code frame starting at offset off. Returns the new offset after the code sequence.
r>c	REF RTOC $55	(off ref -- off')	Transfer the referenced object (words, transitions, variables) from the current process to the morphing code frame starting at offset off. Returns the new offset after the code sequence.

new	NEWCF $78	(init -- offinit=1 cf#)	Allocates a new code frame (from this VM) and returns the code frame number (not CS offset!). If init = 1, then a default (empty) boot and LUT section is created, with sizes based on the current process. The returned offset value points to the next free code address in the morphing code frame.
load	LOAD $71	(cf# ac# --)	Load the code template of agent class AC in the specified code frame number or copy the current code frame (ac=-1). If the template spawns more than one frame, additional frames are allocated and linked.
!cf	SETCF $79	(cf# --)	Switches code morphing engine to new code frame (number). The root frame of the current process can be selected with cf#=-1.
@cf	GETCF $5E	(-- cf#)	Get current code frame number (in CS from this VM).

Code morphing operations

AFL	AML	Stack	Description

`!lut(Δ)`	`SETLUT(Δ)` `$30+Δ`	`( -- )`	Set LUT offset (LP).

`lut [N]`	`LUT N*3 DATA`
	`$7A`
	`DATA`
	`$00`
		`( -- )`	Define a LUT with N rows.

`template AC`	-	`( -- )`	Defines a code frame as a template which is stored in the global CCS and dictionary.
-	`END` `$6E`	`( -- )`	Marks the end of a code frame.
-	`NEXT #CF` `$6F`	`( -- )`	Chains this code frame with a next one.

Code frame control operations

AFL	AML	Stack	Description

`:NAME`	`DEF # S`
	`$74`
	`REFN(n) ..`
		`( -- )`	Defines a new word. A (local) word definition instruction is followed by the LUT row index and the size of the word body. The head of the word body can contain references.

`::NAME`	`DEFN`
	`"NAME"`
	`S`
	`$75`
		`( -- )`	Defines a new word which can be stored in the word dictionary using the export instruction.

`:%NAME`	`TRANS` `$76`	`( -- )`	Defines the transition table.
`sig NAME` `:$NAME`	`DEF # S`	`( -- )`	Defines a signal handler word. The word name must be equal to an already defined signal.

`import NAME try import NAME with name`	`IMPORT # "NAME" $77`	`( -- )`	Import a global word. If the global word does not exists than the agent terminates. To avoid this behaviour, use the try with statement instead, which uses a local word instead.
`export NAME`	`EXT(EXPORT) $06+$20`	`( -- )`	Exports a word to the global dictionary (and *CCS*), which was defined with the ::*NAME* environment.

Code definition and import operations

Stack and Memory Control

AFL	AML	Stack	Description
`dup`	`DUP $68`	`( v -- v v )`	Duplicate the top stack item
`?dup`	`-`	`( v -- 0:v v )`	Duplicate the top stack item only if it is not zero
`drop`	`DROP $6A`	`( v -- )`	Discard the top stack item
`swap`	`SWAP $6B`	`( v1 v2 -- v2 v1 )`	Exchange the top two stack items
`over`	`OVER $6C`	`( v1 v2 -- v1 v2 v1 )`	Make a copy of the second item on the stack
`nip`	`-`	`( v1 v2 -- v2 )`	Discard the second stack item
`tuck`	`-`	`( v1 v2 -- v2 v1 v2 )`	Insert a copy of the top stack item underneath the currents second item
`rot`	`ROT $6D`	`( v1 v2 v3 -- v2 v3 v1 )`	Rotate the positions of the top three stack items

`-rot`	`-`	`( v1 v2 v3 -- v3 v1 v2 )`	Rotate the positions of the top three stack items

`pick(n)`
 	
`pick`
	`FETCH(DS,-n)`
	`$02+(-n)`
	`// PICK $67`
		`( vn .. v1 -- .. v1 vn )`
		`( vn .. v1 n -- .. v1 vn )`
			Get a copy of the n-th data stack item and place it on the top of the stack. (`pick(1)`:`dup`, `pick(0)`: n popped from stack)

`set(n)`
 	
`set`
	`STORE(DS,-n)`
	`$04+(-n)`
	`// SET $66`
		`( vn .. v1 v -- vn* .. v1 )`
		`( vn .. v1 v n -- vn* .. v1 )`
			Modify n-th data stack item. (`set(0)`: n popped from stack!)

`rpick`	`RPICK` `$62`	`R( vn .. v1 -- .. v1)` `( n -- vn )`	Get a copy of the n-th return stack item and place it on the top of the data stack.
`rset`	`RSET` `$63`	`R( vn .. v1 -- vn* .. v1 )` `( v n -- )`	Modify n-th return stack item.
`2dup`	`-`	`( v1 v2 -- v1 v2 v1 v2 )`	Duplicate the top cell-pair of the stack
`2drop`	`-`	`( v1 v2 -- )`	Discard the top-cell pair of the stack
`2swap`	`-`	`( v1 v2 v3 v4 -- v3 v4 v1 v2 )`	Swap the top two cell-pairs on the stack
`2over`	`-`	`( v1 v2 v3 v4 -- v1 v2 v3 v4 v1 v2 )`	Copy cell pair *v1 v2* to the top of the stack
`clear`	`CLEAR` `$5B`	`( .. -- )` `R( .. -- )`	Clear data and return stack.

Data Stack (SS) Operations

AFL	AML	Stack	Description
>R	TOR $64	(v --) R(-- v)	Push the top data stack item to the return stack
R@	-	R(v -- v) (-- v)	Copy the top return stack item to the data stack
R>	FROMR $65	R(v --) (-- v)	Pop the top return stack item to the data stack
2>R	-	(v1 v2 --) R(-- v1 v2)	Push the top data stack cell pair to the return stack
2R@	-	R(v1 v2 -- v1 v2) (-- v1 v2)	Copy the top return stack cell pair to the data stack
2R>	-	R(v1 v2 --) (-- v1 v2)	Pop the top return stack cell pair to the data stack

Return Stack (RS) Operations

AFL	AML	Stack	Description
! !x	STORE/SET $66 STORE(LUT(x)) $04+#	(ref v --) (v --)	Store the value v at memory cell address cf + off retrieved by a lookup in the LUT[ref].
@ @x	FETCH/PICK $67 FETCH(LUT(x)) $02+#	(ref -- v) (-- v)	Fetch and return the value from memory cell address cf + off retrieved by a lookup in the LUT[ref].

`var x typ [ [size] ];`	`-`	`( -- )`	Define and declare a new variable x with a data type typ. Arrays are defined by adding a size operators [size]. Each variable is assigned an AC unique LUT relocation index n.
`const C init`	`-`	`-`	Define a symbolic constant value (def C = init)
`-`	`VAR # S DATA .. $70 DATA $00`	`( -- )` `( -- )`	On VM level a variable is specified by their LUT index # and the size S of the reserved code area in words following the VAR definition.

`-`
 	
`ref(x)`
`%x`
	`REF(#) $08+#`
	
	`VAL(LUT(x))`
		`( -- off cf )`
		
		`( -- # )`
			Inside word bodies or on top-level the REF operator pushes the code offset and code frame of referenced object relocated by the LUT on the data stack.

Data Memory (DS) and Register Operations

Type	Size	Description
integer	word width	Signed Integer
int32	32/(word width) words	Long signed integer
char	Scalar: word width Array: packed 8 bit and (word width / 8) character / word or unpacked (eq. word width)	Character or signed byte

byte	Scalar: word width Array: packed 8 bit and (word width / 8) bytes / word or unpacked (eq. word width)	Unsigned Byte
word	Word width	Unsigned Word
float	2-4 words	Floating Point Number
bool	word width	Boolean

Data types

Mobility

Migration of agents require the update of the boot sections of the code frame and the transfer of the code frame to a neighbour node. Migration to a different VM requires the copying of the code frame.

AFL	AML	Stack	Description
`move`	`MOVE $7F`	`( dx dy -- )`	Migrate agent code to neighbour node in the given direction. The current data and return stack content is transferred and morphed to the boot code section. The transition boot section is loaded with a branch to the current IP+1.
`?link`	`LINK $48`	`( dx dy -- flag )`	Check the link connection status for the given direction. If flag=0 then there is no connection, if flag=1 then the connection is alive.

Migration operations

Process Control

AFL	AML	Stack	Description

|Ai	TCALL(#) $0A+#	(--) R(-- ip cf)	Call next activity word A_i. The word address and the code frame are taken from the *LUT*.
?Ai	TBRANCH(#) $0C+# NOT BRANCHZ(■)	(flag --)	Branch to next transition row for activity A_i if the flag is true. The relative branch displacement for the appropriate TCALL(#) target is first searched by using the *LUT* entry for the respective activity. If this fails, the entire transition section is searched (and the result is cached in the *LUT*). Optimization: replacement with conditional branch, requiring immutable transition section.
.	END $6E	(--)	End marker, which marks the end of a transition table row.

t+ Ai #b t- Ai #b t* Ai #b !t trans	BLMOD $59 TRSET $5A	(ref# b# flag sel --) (ref# --)	Modifies transition table which can be selected by the !t statement. Each transition bound to an outgoing activity is grouped in a block environment. Transition are modified by enabling or disabling the blocks in a transition row using the BLMOD operation. The transition modifiers reference the block number b# in the respective transition row. If b# = 0 all dynamic blocks (in the t-row) will be disabled.
?block	QBLOCK $7B	(flag --)	Suspend code processing if the flag is not zero. If a schedule occurs, the current data and return stack content is transferred and morphed to the boot code section with a branch to the current IP-1.

run	RUN $5F	(arg1 .. #args cf# flag -- id)	Start a new process with code frame (from this VM), returns the identifier of the newly created process. The arguments for the new process are stored in the boot section in the code frame of the new process. If flag = 1 then a forked process is started. The boot section of the new code frame and the boot section of the transition table is modified.
fork	-	(arg1 .. argn #args -- pid)	Fork a child process. The child process leaves immediately the current activity word after forking, the parent process continue.
create	-	(arg1 .. argn #args #ac -- pid)	Create a new agent process.
kill	.. aid=-1: CLEAR EXIT	(aid --)	Terminate and destroy agent. For self destruction the aid must be equal -1.

`suspend`	`SUSP` `$4F`	`( .. flag -- )` `R( .. -- )`	Suspend the execution. The current CF and IP+1 is saved in the current transition table boot section. The process state will be changed to PROC_SUSP. If flag = 1 then the code frame is fully reinitialized after resume and the stacks must be already dumped to the boot section. If flag = -1 then the boot section is initialized and the stacks are dumped, after resume the next instruction is executed directly w/o full code frame setup.

Transition table and scheduling operations

Signals

AFL	AML	Stack	Description
`signal S`	`-`	`( -- )`	Defines a signal *S*.
`:$S .. ;`	`DEF`	`( arg -- )`	Defines a handler for signal *S*. If called, the signal argument is on the top of the data stack. The signal argument is pushed on the data stack.
`raise`	`RAISE` `$49`	`( arg s pid -- )`	Send a signal s to the process *pid*.
`timer`	`TIMER` `$58`	`( sig# tmo -- )`	Install a timer (*tmo*>0) raising signal *sig* if time-out has passed. If *tmo*=0 then remove timer.

Signals

Tuple Database Access

Tuple matching bases on actual parameters ai (values) and formal variable parameters pi (input place holder). The pattern bit-mask M specifies actual (TV,TR) and formal parameters (PR,PS). Database input operations return the values of the formal parameters of the tuple with actual values. The input operations with probe behaviour (try*) will return a status value, too.

```
type TupleArgumentKind = {TV, TR, ANY, PR, PS, MORE }
```

An actual parameter of a tuple is either a value (TV) or a reference to a agent body variable (TR) used to fetch the current value of the variable. A formal parameter of a pattern template tuple is either a wild-card place holder (ANY), a variable reference (PR), or a variable value returned on the stack (PS). The pattern bit-masks can be chained using the MORE value.

AFL	AML	Stack	Description
`out`	`VAL(0) OUT $7C`	`( a1 a2 .. M -- )`	Store a d-ary tuple in the database.
`mark`	`OUT`	`( a1 a2 .. M T -- )`	If t > 0 then a d-ary marking (temporary) tuple with time-out t is stored in the database.
`in`	`VAL(0) IN QBLOCK`	`( a1 a3 .. M -- pi .. p2 )`	Read and remove a tuple from the database. Only parameters are returned. To distinguish actual and formal parameters, a pattern mask p is used (n-th bit=1:n-th tuple element is a value, n-th bit=0: is is a parameter).

tryin	IN $7D	(a1 a3 .. M T -- pi .. p2 0) (a1 a3 .. M T -- a1 a3 .. M T 1)	Try to read and remove a tuple. The parameter t specifies a time-out. If t = -1 then the operation is non-blocking. If t = 0 then the behaviour is equal to the in operation. If there is no matching tuple, the original pattern is returned with a status 1 on the top of the data stack, which can be used by a following ?block statement. Otherwise a status 0 is returned and the consumed tuple.
rd	VAL(0) RD QBLOCK	(a1 a3 .. M -- pi .. p2)	Read a tuple from the database. Only parameters are returned.
tryrd	RD $7E	(a1 a3 .. M T -- pi .. p2 0) (a1 a3 .. M T -- a1 a3 .. M T 1)	Try to read a tuple from the database. Only parameters are returned. Same behaviour as the tryin operation.
?exist	VAL(-2) RD	(a1 a3 .. M -- 0|1)	Check for the availability of a tuple. Returns 1 if the tuple does exist, otherwise 0. Is processed with a RD/tryrd and t=-2.
rm	VAL(-2) IN	(a1 a2 .. M --)	Remove tuples matching the pattern.

Tuple space operations based on pattern matching

```
( enum TupleArgumentKind [1] TV TR ANY PR PS MORE ;  Built-in
Type!!! )

APL: out(SENSOR,100,10);
AFL: SENSOR 100 10 0o{TV,TV,TV} out
```

```
AML: 47 100 10 0o111 0tupletime OUT

APL: in(SENSOR,?x,?y)
AFL: SENSOR 0o{TV,PS,PS} in x ! y !
AML: 47 0o155 0tmo IN REF(x) STORE REF(y) STORE

APL: in(SENSOR,?x,10)
AFL: SENSOR 10 ref(x) 0o{TV,PR,TV} in
AML  47 10 2LUTIND 0o141 0tmo IN

APL: stat:=try_in(1000,SENSOR,?x,10)
AFL: SENSOR 10 0o{TV,PS,TV} 1000 tryin if x ! then
AML: 47 10 0o151 1000 IN <status?> REF(x) STORE

APL: exist?(SENSOR,?,10)
AFL: SENSOR 10 0o{TV,ANY,TV} ?exist
AML: 47 10 0o131 -2exist RD <status?>
```

Examples of tuple space operations and usage of the pattern mask

Examples

```
Some Code
```

Version

Version: 1.4.3
Revision: Change of tuple space operations semantics, introduction of value constructors (octal, hexadecimal)
Author: Stefan Bosse

See Also

AVM Architecture
