Table of Contents

AFL

Description

AFL is the Agent Forth Programming Language (αFORTH). There is a machine instruction subset AML (Agent Forth Machine Language) that can be directly executed by the Agent Forth Virtual Machine (AFVM). Agent Forth consists of common stack-based Forth operations and special agent-related instructions.

Content

AFL Overview and Introduction

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
 ( $v_1$ $v_2$ $v_3$ -- $r_1$ $r_2$ $r_3$ )
Return Stack
R( $v_1$ $v_2$ $v_3$ -- $r_1$ $r_2$ $r_3$ )
```

The top of the stack is the right underlined element of the group (i.e., r_3). 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-initialization 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 Layout

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

AFL	AML	Description
`:*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 optimizing 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)

Code and Call Frames

The following terms 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.

Data Types

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

AFL Data & Math Operations

Name

AFL Instruction Set: Data and Mathematical Operations

Synopsis

Paragraph text...

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..2^{n-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 {-2^{n-1}..-2^{n-2}-1} or {2^{n-2}..2^{n-1}-1} with n bit data width.

`0/1`	`ZERO/ONE` `$40/$41`	`( -- v )`	Pushes value zero or one to the data stack.

AFL Value	AML	Stack	Description
`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`	(v_1 v_2 -- r)	Addition (signed) r = $v1+v_2$
`-`	`SUB` `$43`	(v_1 v_2 -- r)	Subtraction (signed) r = v_1-v_2
`*`	`MUL` `$44`	(v_1 v_2 -- r)	Multiplication (signed) r = v_1*v_2
`/`	`DIV` `$45`	(v_1 v_2 -- r)	Division (signed) r = $v1/v_2$
`mod`	`MOD` `$46`	(v_1 v_2 -- r)	Division (signed) returning remainder r = v_1 % v_2
`negate`	`NEG` `$47`	(v -- r)	Negate operand r=-v
`abs`	-	(v -- r)	Return positive equivalent r= if $v_1<0$ then $-v_1$ else v_1
`min`	-	(v_1 v_2 -- r)	Return smallest number r= if $v_1<v_2$ then v_1 else v_2
`max`	-	(v_1 v_2 -- r)	Return biggest number r= if $v_1>v_2$ then v_1 else v_2
`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	(v_1 v_2 -- r)	Lower than (signed) r = (v_1<v_2)
>	GT $4B	(v_1 v_2 -- r)	Greater than (signed) r = (v_1>v_2)
=	EQ $4C	(v_1 v_2 -- r)	Equal (signed) r = (v_1=v_2)
<>	-	(v_1 v_2 -- r)	Not equal (signed) r = (v_1<>v_2)
<=	LE $4D	(v_1 v_2 -- r)	Lower than or equal r = (v_1<=v_2)
>=	GE $4E	(v_1 v_2 -- r)	Greater than or equal r = (v_1>=v_2)
0=	-	(v -- r)	Test for zero
0<>	-	(v -- r)	Test for non zero
within	-	(v_1 v_2 v_3 -- r)	Test if v_2 is within [v_1,v_3]

Relational Operations

AFL Operator	AML	Stack	Description
and	AND $50	(v_1 v_2 -- r)	$r = v_1 \wedge v_2$
or	OR $51	(v_1 v_2 -- r)	$r = v_1 \vee v_2$
xor	-	(v_1 v_2 -- r)	r = v_1 xor v_2
not	NOT $52	(v -- r)	$r = \neg v$
invert	INV $53	(v -- r)	$r = \forall \text{bits } b_i(v): \neg b_i$
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 bit-wise 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`	(v_1 v_2 -- v_2 v_1)	Exhange the top two stack items
`over`	`OVER` `$6C`	(v_1 v_2 -- v_1 v_2 v_1)	Make a copy of the second item on the stack
`nip`	-	(v_1 v_2 -- v2)	Discard the second stack item
`tuck`	-	(v_1 v_2 -- v2 v1 v2)	Insert a copy of the top stack item underneath the currents second item
`rot`	`ROT` `$6D`	(v_1 v_2 v_3 -- v_2 v_3 v_1)	Rotate the positions of the top three stack items
`-rot`	-	(v_1 v_2 v_3 -- v_3 v_1 v_2)	Rotate the positions of the top three stack items

`pick(n)`
`pick`
	`FETCH(DS,-n)`
	`$02+(-n)`
	`// PICK $67`
		(v_n .. v_1 -- .. v_1 v_n)
		(v_n .. v_1 n -- .. v_1 v_n)
			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 from stack)

`set(n)`
`set`
	`STORE(DS,-n)`
	`$04+(-n)`
	`// SET $66`
		(v_n .. v_1 v -- v_n* .. v_1)
		(v_n .. v_1 v n -- v_n* .. v_1)
			Modify n-th data stack item. (`set(0)`: n taken from stack!)

AFL	AML	Stack	Description
rpick	RPICK $62	R(v_n .. v_1 -- .. v_1) (n -- v_n)	Get a copy of the n-th return stack item and place it on the top of the data stack.
rset	RSET $63	R(v_n .. v_1 -- v_n^* .. v_1) (v n --)	Modify n-th return stack item.
2dup	-	(v_1 v_2 -- v_1 v_2 v_1 v_2)	Duplicate the top cell-pair of the stack
2drop	-	(v_1 v_2 --)	Discard the top-cell pair of the stack
2swap	-	(v_1 v_2 v_3 v_4 -- v_3 v_4 v_1 v_2)	Swap the top two cell-pairs on the stack
2over	-	(v_1 v_2 v_3 v_4 -- v_1 v_2 v_3 v_4 v_1 v_2)	Copy cell pair v_1 v_2 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	-	(v_1 v_2 --) R(-- v_1 v_2)	Push the top data stack cell pair to the return stack
2R@	-	R(v_1 v_2 -- v_1 v2) (-- v_1 v_2)	Copy the top return stack cell pair to the data stack

AFL	AML	Stack	Description

`2R>`	-	`R( v`$_1$` v`$_2$` -- )`
		`( -- v`$_1$` v`$_2$` )`
			Pop the top return stack ce

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 Types

Examples

```
Code
Code
```

Authors

Paragraph text...

See Also

Paragraph text...

AFL Control Operations

Name

AFL Control Flow Operations

Synopsis

```
if else then
case of end of endcase
exit
do loop
do loop+
begin again
begin until
begin while repeat
```

Description

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

Branches

There are boolean conditional and expression multi-value branches. The conditional parameters are stored on the data stack. Commonly the high-level control structures are transformed in low-level flow control supported by AML (relative branches). The call instruction stores the current *IP/CF* pair on the return stack before the control flow is branched to the user defined function word.

AFL	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.

AFL	AML	Stack	Description

`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 {v_1, v_2, ..}. If a value matches the key, the words between `of` and `endof` are executed.

`exit`	`EXIT`
	`$60`
		`R( ip cf -- )`
		`( -- ` r_1 r_2 `.. )`
			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).

word	`CALL(#)`
	`$0E+#`
		`R( -- ip cf )`	Call a word and push the current code position *IP*+1 on the return stack and the

AFL	AML	Stack	Description
			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 (ΔS≥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 placeholder)!

Branch control structures

Loops

There are counting and conditional loops. The loop parameters and conditional values are stored on the data stack.

AFL	AML	Stack	Description

`do .. loop`
`do .. loop+`
	-	`( limit index`$_0$` -- )`
		`( limit index`$_0$` -- )`
		`( 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 inner moss first-level loop is referenced with `i`, and outer higher level loops with `j` and `k`.

Loop control structures

Examples

```
Code
Code
```

Authors

Stefan Bosse

See Also

AFL Overview

AFL Code Frames

Name

AFL Code Frame Definition, Control, and Modification

Synopsis

```
:NAME
::NAME
:%NAME
sig NAME :$NAME
import NAME try import NAME with NAME export NAME
c> v>c >c s>c r>c new load !cf @cf
```

Code definition and import operations

AFL	AML	Stack	Description

`:`*NAME*	`DEF # `*Sn*
	`$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 # `*Sn*	`( -- )`	Defines a signal handler word. The word name must be equal to an already defined signal.

AFL	AML	Stack	Description

`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` 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 morphing operations

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.

AFL	AML	Stack	Description
`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 --` $off^{init=1}$ `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.

AFL	AML	Stack	Description
`@cf`	`GETCF` `$5E`	`( -- cf# )`	Get current code frame number (in *CS* from this VM).

Code frame control 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.

Examples

```
Code
Code
```

Authors

Stefan Bosse

See Also

AFL Overview

AFL Mobility

Name

AFL Code Frame Mobility Instructions

Synopsis

```
move
?link
```

Description

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.

A snapshot is created before code frame migration by dumping the stack content (from the data and return stack) to the primary boot section and by modifying the secondary boot section on the transitions definition word storing the code entry point after migration.

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

Examples

```
Code
Code
```

Authors

Stefan Bosse

See Also

AFL Overview

AFL Process Control

Name

AFL Process and Agent Control including Reconfiguration

Synopsis

Paragraph text...

Description

Agent processes can be created at run-time by any other agent. An agent process can be created from a template or forked from an existing parent agent (composing parent-child groups). Each agent process can created with a distinct set of parameter arguments. In the case of process forking, a copy and snapshot of the current code frame is created. Both processes will continue execution with the same data and control state except the process arguments.

Transition Definition

The transition definition table section consists of rows, each starting with a specific activity word and a sequence of conditionalö activity branching instructions.

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*.

AFL	AML	Stack	Description

`?Ai`	`TBRANCH(#)`
	`$0C+#`
	`NOT BRANCHZ(Δ)`
		`( 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.

Transition Definition Modification

AFL	AML	Stack	Description

`t+ A`$_i$ *#b*
`t- A`$_i$ *#b*
`t* A`$_i$ *#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.

Process Control

AFL	AML	Stack	Description
`?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.

AFL	AML	Stack	Description
`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 acitivity 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

AFL	AML	Stack	Description
			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.

Examples

```
Code
Code
```

Authors

Stefan Bosse

See Also

AFL Overview

AFL Communication

Name

AFL Agent Interaction and Communication

Synopsis

```
signal S :$S raise timer
out mark in rd tryin tryrd rm
```

Description

Agents can communicate with each other by two methods: (1) Signals; (2) Tuples. Signals are simple messages that can be propagated in the network. they are primarily used for parent-child agent interaction. Tuples can only be exchanged by agents located on the same node by accessing a tuple database. Signals are asynchronously processed, thereby tuple access can be handled synchronously on the consumer side by agent blocking.

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 timeout has passed. If *tmo*=0 then remove timer.

Tuple Spaces

A tuple is basically an ordered list of values. The single values can have different data types, in contrast to mono-typed arrays. A tuple has a dimension like a vector determining the number of elements. Tuple space access is generative, i.e., there are producer and consumer (processes / agents), and a tuple stored in the tuple database by a producer can still exist after the producer process (agent) had terminated.

Tuple matching bases on actual parameters a i (values) and formal variable parameters p i (input place holder). The pattern bit-mask *M* specifies actual (`TV` , `TR`) and formal parameters (`TR` , `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 placeholder (`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 timeout *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).

AFL	AML	Stack	Description
`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 timeout. 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 Operations

```
enum TupleArgumentKind [ 1 ] TV TR ANY PR PS MORE ;

APL : out( SENSOR , 100 , 10 );
```

```
AFL : SENSOR 100 10 0o{ TV , TV , TV } out
AML : 47 100 10 0o111 0^tupletime OUT

APL : in ( SENSOR , ?x , ?y )
AFL : SENSOR 0o{ TV , PS , PS } in x ! y !
AML : 47 0o155 0^tmo 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 2^LUTIND 0o141 0^tmo 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 -2^exist RD status?
```

Examples of tuple space operations and usage of the pattern masks

Examples

```
Code
Code
```

Authors

Stefan Bosse

See Also

AFL Overview

DOS

Description

The Distributed Operating System layer is used to compose a virtual communicating machine of heterogeneous network nodes, mainly Internet and Intranet connected computers and servers.

Content

DOS RPC

Name

Remote Procedure Call API (RPC)

Synopsis

```
JavaScript DOS
Module RPC
```

Description

The Remote Procedure Call API (RPC) is used for point-to-point communication between processes, nodes, and Browser applications.

Content

DOS RPC: RPC Message Handler

Module

JavaScript Rpc

Object Interface

```
enumeration Operation = {
    SEND
    RECV
    TRANSREQ
    TRANSREP
    TRANSAWAIT
    TRANS
    GETREQ
    PUTREP
    IAMHERE
    WHOIS
    LOOKUP
}

constructor Rpcio (operation?
    hdr?: Network.header
    data?: {Buffer|string}
    context?: Scheduler.taskcontext
    callback?: function) →
rpcio :
{
    RPC Operation
    operation :Operation
    Public server port (used for GETREQ only)
    pubport :Network.port
    Connection Link port (used by VLC only, incoming link)
    connport :Network.port
    RPC transaction header
    header :Network.header
    RPC data buffer
    data :Buffer
    pos :number
    Process Context
    context :Scheduler.taskcontext
    Reply callback (local RPC only)
    callback : function(rpcio)
    Source host port - From (Request: Caller, Reply: Executor)
    hostport :Network.port
    Destination host port - To (Request: Executor, Reply: Caller)
    sendport :port
    Message Forwarding
    hop :number
    hop_max :number
    Transaction Identifier
    tid :number
    Timeout Garbage Management
    timeout: number
    Overall Status of the RPC operation
    status :Network.Status
    Packet Pool Index
    index :number
}
```

Object Methods

```
rpcio.init = function(operation,hdr,data,context,callback)
rpcio.to_xml = function(wrap) → body:string
rpcio.of_xml = function(xml) → status:number
```

Description

Each Remote Procedure Call (**RPC**) operation is handled with a *RPCIO* object. It contains the RPC transaction header and optional data. *RPCIO* handlers are also used for server localization. A *RPCIO* handler can be forwarded to a broker server or to another node host. The host and send ports are used for *RPCIO* forwarding. The host port is source of a *RPCIO* message (the host the message is coming from), and the send port is the destination host port sending the message to.

Examples

```
Code
Code
```

Authors

Stefan Bosse

See Also

Paragraph text...

DOS RPC: RPC Client-Server API

Module

JavaScript Rpc

Object Interface

```
constructor RpcInt (router:rpcrouter) →
rpcint : {
    router :rpcrouter
    transaction_id :number
}
```

Object Methods

```
rpcint.trans = function(rpcio:rpcio,callback:function(stat:Net.Status))
rpcint.getreq = function(rpcio:rpcio)
rpcint.putrep = function(rpcio:rpcio)
```

Description

A server process executes the `getreq` operation by supplying the private server port in the `rpcio.header` field. The public server port required by the router is computed from private port (using a port mapping cache). The server request listening is handled by the RPC router. A server process executing the `getreq` operation will be blocked until a matching client request arrives. There can be multiple server processes listening on the same server port.

A client send a transaction to the server by providing the public server port in the `rpcio.header` field.and using the `trans` operation. If the reply of the transaction arrives (or the RPC failed), the provided callback function is executed with the status of the operation. The reply is contained in the original *RPCIO* handler.

The *RPCIO* message is forwarded to the respective server host and passed finally to the server process, waking up the process. After the server has serviced the request, it responds with a reply passed by the `putrep` operation.

Examples

```
var privhostport = Net.uniqport();
var pubhostport = Net.prv2pub(privhostport);
var scheduler = Sch.TaskScheduler();
var router = Router.RpcRouter(pubhostport);
var rpc = Rpc.RpcInt(router);
var privsrvport = Net.port_of_str('12:34:56:78:89:01');
var pubsrvport = Net.prv2pub(privsrvport);

var dying = false;
var rpcio = Rpc.Rpcio();

Sch.ScheduleLoop(
    function () {return !dying;},
    [
        function () {
            rpcio.init();
            rpcio.header.h_port = privhostport ;
            rpc.geteq(rpcio);
        },
        function () {
            ... service request ...
        },
        function () {
            rpc.putrep(rpcio);
        }
    ]);
```

A sample RPC Server

```
var privhostport = Net.uniqport();
var pubhostport = Net.prv2pub(privhostport);
var cap = Net.Capability(pubhostport);

var scheduler = Sch.TaskScheduler();
var router = Router.RpcRouter(pubhostport);
var rpc = Rpc.RpcInt(router);
var privsrvport = Net.port_of_str('12:34:56:78:89:01');
var pubsrvport = Net.prv2pub(privsrvport);
 ...
var stat = Status.STD_UNKNOWN;
var info = '';

Sch.ScheduleBlock([
    function () {
        var rpcio = Rpc.Rpcio();
        rpcio.header.h_port = cap.cap_port;
        // rpcio.header.h_priv = cap.cap_priv;
        rpcio.header.h_command=Command.STD_INFO;
```

```
            rpc.trans(rpcio,function (_stat) {
                stat = _stat;
                if (stat==Status.STD_OK) info=Buf.buf_get_str(rpcio);
            });
         },
        function () {
         ...
        }
   ])
```

A sample RPC Client

Author

Stefan Bosse

See Also

DOS Scheduler

Name

JavaScript Task Scheduler

Synopsis

```
JavaScript
Module Scheduler
```

Description

JavaScript is strictly single threaded. Though timeout callbacks can be used to execute function apparently concurrently, there is no concept of task blocking. But most programming models, for example, communication, relies on a concurrent behaviour.

To avoid deep nesting of asynchronous callback functions, a task scheduler was invented. A task supports virtualized blocking handled by a scheduler. A common multi-threaded program flow can be implemented by using the task scheduler and task contexts (Task Context), though there is still no concurrent execution or preemption.

- The Scheduler module enables programming of scheduled multi-process programs compatible with all Browsers and node VMs.
- The scheduler offers linear programming of blocking statements.
- The scheduler offers the implementations of Activity-Transition Graphs (ATG)
- Timer handler can be created and started like any other procedural action.

Content

DOS Scheduler: Task Context

Module

JavaScript Scheduler

Object Interface

```
constructor TaskContext (id:number,obj?:object) →
taskcontext : {
    Context Identifier Number
    id : number
    Context Blocking
    blocked : boolean
    Transition functions
    trans : []
    Scheduling Blocks
    block : []
    ATG Object
    obj :object
    Timeout Management
    timeout :number
    Timer Handler
    timer :number
    Current activity
    state : function()
}
```

Object Methods

```
None
```

Description

A task context is a virtual process context that is executed by the scheduler. A task context consists of function blocks, each consisting of nameless functions. The task context supports blocking. An nameless function of a task block may block (calling a function setting the blocked property of the task object to false) at the end of the program flow of the function body. The next function in a task block (or the next function block) is called only if the task is unblocked (blocked =false).

Furthermore, a task context can be used to implement Activity-Transition Graphs (ATG). An ATG object consists of named activity functions and a transition function array. An ATG object (passed to the TaskContext constructor by the proc parameter) must have a transitions method returning an array that defines the activity transitions.

The currently executed activity function is stored in the state property of the task object.

ATG Object

```
var atg = function() {
    var cx,cy,cz,..;
    this.init = function () {...};
    this.act1 = function () {...};
    this.act2 = function () {...};
    this.act3 = function () {...};
     ...
    this.transitions = function () {
        return [
            [undefined, this.init, function (con) {return true)],
            [this.init, this.act1, function (con) {return ε(cx)],
            [act_i , act_j , cond_ij ], conditional transition
            [act_i , act_j ], unconditional transition
             ...
    }
}
```

Each row of the transitions array consists of three elements:

1. The outgoing activity function;
2. The next activity function;
3. The transition function returning a Boolean value. Only if the value is true and the blocked property of the task object is false than the transition is executed.

An activity function may block at the end of the program flow (setting the blocked property of the task object to true).

Examples

```
var Sch = require('scheduler');
var rpcserver = function(rpc,pubport, privport ) {
  var main=this;
  var dying = false;
  this. privport = privport;
  this.thread = function (arg) {
    var thr=this;
    var rpcio = Rpc.Rpcio();

    this.init = function () {
    };
    this.req = function () {
      rpcio.init();
      rpcio.operation = Rpc.Operation.GETREQ;
      rpcio.header.h_port = privport;
      rpcio.header.h_status = undefined;
      rpcio.header.h_command = undefined;
      rpcio.header.h_priv = undefined;
      rpc.getreq(rpcio);  blocking operation
    };
    this.service = function () {
      service request in rpcio
    };
    this.reply = function () {
      rpc.putrep(rpcio);
    };
    this.terminate = function () {
      do something
    };
    this.transitions = function () {
      return [
        [undefined,init],
        [init,req],
        [req,service],
        [service,reply],
        [reply,req,function () {return !dying;}],
        [reply,terminate,function (){return dying;}]
      ];
    };
    this.context=Sch.TaskContext('myserver'+arg,thr);
  };
}
```

A simple RPC server loop

Authors

Stefan Bosse

See Also

Task Functions
Task Scheduler

DOS Scheduler: Task Scheduler

Module

JavaScript Scheduler

Object Interface

```
constructor TaskScheduler() →
taskscheduler: {
    List of all process context objects
    context:taskcontext []
    List of all current callback blocks (can be empty)
    callbacks: function []
    List of all (timer) handler objects (can be empty)
    handler:function []
    Currently executed task context
    current:taskcontext
    nextid:number
    lock:number
    nested:number
    If reschedule > 0, the scheduler is executed
    immediately (ASAP),
    but without preemption of the current process task.
    reschedule:number
}
```

Object Methods

```
taskscheduler.add_callback =
  function(callback:{function|[function,..]})
taskscheduler.add_timer =
  function(timeout,name,callback,once)
taskscheduler.remove_timer =
  function(name)
taskscheduler.Add =
  function(con:taskcontext)
taskscheduler.Init =
  function()
taskscheduler.Schedule =
  function()
taskscheduler.Run =
  function()
```

Description

The object methods are usually not invoked directly. Instead there is a set of function operating on the current scheduler object (commonly there is only one scheduler). The Scheduler module implements virtual processes based on scheduling blocks (SB) and Activity-Transition graphs (ATG) supporting virtual process blocking and simulated multi-tasking.

add_callback

Add a callback function executed once in the next scheduler run before any process (context) activity execution. The callback argument is either a function, or [*function*] or [*function* , arg_1 , arg_2 ,.., arg_9] array.

add_timer

Add a timer handler function (callback) executed once or continuously by the scheduler. Timers are scheduled before any context processes are scheduled. A timer handler is executed in its own context. Therefore, the handler function may create scheduling blocks containing blocking statements. The timeout is specified in millisecond units and is affected by the tick resolution of the scheduler (commonly 10 ms).

remove_timer

Remove a timer handler identified by its name.

Examples

```
var scheduler = Sch.TaskScheduler();
scheduler.Init();
```

Authors

Stefan Bosse

See Also

Task Functions
Task Context

DOS Scheduler: Task Functions

Module

JavaScript Scheduler

Functions

```
function AddTimer( timeout,name,callback,once)
function Bind(obj:object,method:function) → object
function Delay(millisec)
function exec_block_fun(
  next:{function|[function]|[function,arg1,arg2,..]})
function FunContext(
  sched:{taskscheduler|undefined},
  id,
  fun,
  arg?)
function GetId() → id:number
function GetCurrent() → taskcontext
function GetScheduler() → taskscheduler
function GetTime() → ticks:number
function IsBlocked(context?) → blocked:boolean
function ScheduleCallback(callback:function)
function ScheduleBlock(
  block,
  handler:function)
function ScheduleLoop(
  cond:function,
  body:function [],
  finalize:function [],
  handler:function)
function ScheduleNext()
function SetBlocked(blocked,context?)
function Suspend(context?)
function Wakeup(context)
```

Description

The Scheduler module implements virtual processes based on scheduling blocks (SB) and Activity-Transition graphs (ATG) supporting virtual process blocking and simulated multi-tasking.

Scheduling blocks are created with the ScheduleBlock and ScheduleLoop functions. They add a scheduling sequence to the current context consisting of functions scheduled ASAP. The loop block repeats the execution of the loop body block (`body`) as long as the iteration function (`cond`) returns a true value (checked before the loop body is scheduled). A scheduling block is executed after the current task is suspended (or ends).

Each scheduling block can define an optional exception handler function executed if an exception occurred inside a scheduling block. The loop scheduling block can define an optional finalizing function that is executed after the last loop iteration.

A function of a scheduling block may add more scheduling blocks or loops, which are added on the top of the current scheduling block, therefore, executed after the current function terminates!

AddTimer

Add a timer handler function (`callback`) executed once or continuously by the scheduler. Timers are scheduled before any context processes are scheduled. A timer handler is executed in its own context. Therefore, the handler function may create scheduling blocks containing blocking statements. The timeout is specified in millisecond units and is affected by the tick resolution of the scheduler (commonly 10 ms).

FunContext

Create and add a new functional task context to the given scheduler (if undefined the current scheduler is used). Inside the function scheduling blocks and scheduling loops can be used.

GetCurrent

Return current process context object.

GetTime

Get current system time in tick units.

ScheduleCallback

Schedule an asynchronous callback function execution. Must be preemption-save! I.e., if the program is currently within a scheduling action, the callback is queued, otherwise it will be executed immediately.

ScheduleNext

Call the scheduler ASAP, eventually with a callback function executed before any ready process block.

Suspend

Set the blocked attribute of the current or specific context.

Wakeup

Reset the blocked attribute of a specific context.

Examples

```
Code
Code
```

Authors

Stefan Bosse

See Also

Task Context
Task Scheduler

DOS Scheduler: Mutex Lock

Module

JavaScript Scheduler

Object Interface

```
constructor Lock() →
lock: {
    Mutex Lock State
    locked: boolean
    Blocked and waiting processes
    waiter: taskcontext []
    Current Owner Process
    owner: {taskcontext|undefined}
 }
```

Object Methods

```
lock.init=function()
lock.acquire=function()
lock.try_acquire=function() → boolean
lock.release=function()
lock.is_locked=function() → boolean
```

Description

Inter-process synchronization and Mutual Exclusion Lock Object that is used to protect critical code sections. A lock object may only be used (i.e., acquired) in scheduling blocks (last statement of a block element)!

init

Initialize the Mutex Lock Object.

acquire

Lock the Mutex Lock Object. If the Lock is already acquired by another process, the current context process is suspend and queued in the waiter list of the Lock object. If the owner of the Lock releases the Lock, the next waiting process is scheduled.

try_acquire

Try to acquire a Lock. If the Lock is free, the operation returns a Boolean true value, otherwise a false value. This operation is non-blocking.

release

Unlock the Mutex Lock Object. If there are waiting processes, schedule the next process.

Examples

```
Code
Code
```

Authors

Stefan Bosse

See Also

Task Context
Task Scheduler

Index