R A M F O R T H

R E F E R E N C E   M A N U A L

FOR ATMEL AVR MICROCONTROLLERS
VERSION Ø.9.Ø

MATT SARNOFF
OCTOBER 29, 2019

BSD 3-CLAUSE LICENSE

October 29, 2019
Typeset on a Smith-Corona Coronet XL. and it's full of mistakes!!!!
(can you spot them all???)
twitter.com/txsector
msarnoff.org
github.com/74hc595

TABLE OF CONTENTS

I. INTRODUCTION

Ramforth is a token-threaded (i.e. bytecode-interpreted) implementation of the Forth programming language for 8-bit Atmel AVR microcontrollers, specifically the ATmega1284/ATmega1284p, which have 16KB of RAM and 128KB of flash ROM. It has been designed to suit the constraints of a 16KB address space. It was also designed for interactive development; rather than compiling Forth to AVR machine code to be executed from flash (and reducing the the lifespan of the chip), Forth is compiled into a compact bytecode, which can occupy as little as half the space of code generated by other 8-bit Forths. To maximize use of the (quite small) amount of) memory, it is possible to free up all the space used for metadata and compile-time code (in Forth terminology, the "heads" of dictionary entries) allowing nearly all the device's memory to be used for code, data, and screen buffers.

Interaction with the Ramforth interpreter can be done via serial port, but it has been designed for the "Amethyst" computer, which gives the AVR access to video and audio outputs, keyboard input, and peripherals such as gamepads and SPI memory devices like EEPROMs, FeRAMs, and SD cards. Ramforth thus includes support for 40x25 and 80x25 text modes supported by the Amethyst, a full-screen editor, and support for all the Amethyst's bitmap graphics modes.

Ramforth includes over 700 words, including all words in the Forth 2012 standard core, core ext, double, double ext, exception, string, and tools word sets. (There are a few exceptions to standard Forth, which are noted in this manual.)

This is a hobby project. If you are looking for a real Forth for AVR, consider Amforth (amforth.sourceforge.net) or FlashForth (flashforth.com). Finally, please note that this is a reference manual, not a Forth tutorial. The de facto guides for learning Forth are Leo Brodie's Starting Forth and Thinking Forth. Threaded Interpretive Languages by R. G. Loeliger is an excellent resource for diving into Forth's low-level implementation details.

II. MEMORY LAYOUT AND DICTIONARY STRUCTURE

In the vast majority of Forth systems, the dictionary is treated as one contiguous block of memory, where code, data, names, links, etc. are all interspersed. Due to the limited working memory of the AVR, a key design goal was the ability to delete information only needed at compile time (word names, and words that extend the compiler/interpreter) without affecting the compiled bytecode. In fact, Ramforth supports a "tiny" mode, where the compiler/interpreter release all their memory (dictionary, input buffers, pictured numeric output buffers, etc.) to be utilized by applications. In tiny mode, the data and return stacks are reduced to 8 entries each, leaving the remaining 16,320 bytes free for code, data, variables, and screen buffers. (Note that the Amethyst system reserves the topmost 32 bytes of RAM for its own purposes.)

The following is a memory map of the Ramforth system (non-tiny mode):

\$0000-\$00FF	AVR hardware registers
\$0100-\$3FCD	Free memory
\$3FCE-\$400F	Data stack
\$4010-\$4047	Forth runtime variables
\$4048-\$4097	Terminal input buffer
\$4098-\$40DF	Return stack
\$40E0-\$40FF	System reserved memory

When used with a video screen, the uppermost part of the "free memory" area is used as a screen buffer. The amount of space occupied by the buffer depends on video mode:

40x25 mono text mode	1000 bytes
40x25 color text mode 80x25 mono text mode	2000 bytes
Bitmap modes	2000-16000 bytes

The "free memory" region is then divided into <u>code space</u> and <u>name space</u>. Code space starts at \$0100 and grows upward. It contains the "bodies" of most words; executable bytecode, variables, arrays, etc. Name space initially starts at \$0500, but it movesupward to allow code space to grow as necessary. Name space contains a linked list of all user-defined words, i.e. your typical Forth dictionary. Each dictionary entry in name space specifies: the location of the previous entry specified as an 8-bit offset (thus, dictionary entries can be no longer than 255 bytes), the word's name and its length (up to 31 characters), and either a pointer to the word's body in code space (colon-words, VARIABLEs, etc.) or the body itself inline (CONSTANTs and user-defined compiling words). Words with their bodies inline in name space are called "compiling words" or "compiler words." They contain information that is only needed at compile time, and any words the user has defined to alter the behavior of the compiler/interpreter. Code in code space cannot reference code or other data in name space. Code in name space may call code elsewhere in name space or in code space. Name space may be located anywhere in memory; code in code space cannot rely on name space beginning at any particular address. Furthermore, all references from name space to another location in name space are relative offsets.

III. SYNTAX

Ramforth follows traditional Forth syntax. Forth code is a sequence of words separated by spaces. When the interpreter encounters a word, it looks it up in the dictionary and performs the word's execution semantics. If the word is not found in the dictionary, it is parsed as a number, and the appropriate numeric value is pushed on the stack. If it is not a valid number, the interpreter reports an "undefined word" exception.

All words in Forth are case-insensitive. Word names may be up to 31 ASCII characters in length. They may contain any character. (though you'll have trouble trying to execute words with spaces or newlines in them from the interpreter!)

If a number does not have a prefix or suffix, it is interpreted as a value in the current numeric base, as reported by BASE C@. The $ prefix causes the value to be interpreted as hexadecimal regardless of the current base; i.e. $7A69 will always be interpreted as the value 31337_{10}. The % prefix causes the number to be interpreted as binary, e.g. %10101010 evaluates to 170_{10}. The # prefixes forces the value to be interpreted in decimal.

A number ending in a period (e.g. 12345678.) is interpreted as a double-cell value. (32 bits) Otherwise, the number is interpreted as a 16-bit value. (the size of a single cell.)

A single character between single quotes (e.g. 'a') pushes the ASCII value of that character.

A string of two numbers separated by a caret ^ without any space between, e.g. 123^74, is interpreted as a byte pair, resulting in a 16-bit cell value with the low byte equal to 123 and the high byte to 74, i.e. $4A7B. Ramforth uses byte pairs to represent xy coordinates, since most video modes do not have dimensions that exceed 255 pixels. This optimization eliminates the extra overhead of 16-bit math when 8-bit arithmetic suffices.

IV. WORD LISTS

Words whose behavior conforms to the Forth 2012 specification are listed without further explanation. Deviations from the specification and behavior of words exclusive to Ramforth are mentioned as needed.

Summary of stack effect notation:

x	any cell-sized (16-bit) value
n	signed integer (16-bit)
u	unsigned integer (16-bit)
d	signed double-cell integer (32-bit)
ud	unsigned double-cell integer (32-bit)
flag	cell-sized boolean value; $0000=false $FFFF=true
w h c	character (8-bit)
c-addr	address of an 8-bit value or sequence of 8-bit values
a-addr	address of a cell-sized (16-bit) value or sequence of cell-sized values
xy wh p	byte pair
"(char1)ccc(char2)"	consumes delimiters char1 from input stream, accepts characters ccc from input stream, then consumes delimiters char2 from input stream
xt	execution token
nt	name token

IV.1. Comment words

(	("ccc(paren)" --)	ignore up to next right-paren
\	("ccc(eol)" --)	ignore rest of line

IV.2. Stack operators

-ROT	(x_1 x_2 x_3 -- x_3 x_1 x_2)	
?DUP	(x -- 0 / x x)	
>R	(x --) (R: -- x)	
2>R	(x_1 x_2 --) (R: x_1 x_2)	
2DROP	(x_1 x_2 --)	
2DUP	(x_1 x_2 -- x_1 x_2 x_1 x_2)	
2OVER	(x_1 x_2 x_3 x_4 -- x_1 x_2 x_3 x_4 x_1 x_2)	
2R@	(-- x_1 x_2) (R: x_1 x_2 -- x_1 x_2)	
2R>	(-- x_1 x_2) (R: x_1 x_2 --)	
2RDROP	(R: x_1 x_2 --)	equivalent to <u>UNLOOP</u>
2ROT	(x_1 x_2 x_3 x_4 x_5 x_6 -- x_3 x_4 x_5 x_6 x_1 x_2)	
2SWAP	(x_1 x_2 x_3 x_4 -- x_3 x_4 x_1 x_2)	
DEPTH	(-- +n)	
DROP	(x --)	
DUP	(x -- x x)	
NIP	(x_1 x_2 -- x_2)	
OVER	(x_1 x_2 -- x_1 x_2 x_1)	
PICK	($x_u \ldots x_1$ x_0 u -- $x_u \ldots x_1$ x_0 x_u)	
R@	(-- x) (R: x -- x)	
R>	(-- x) (R: x --)	
RDROP	(--) (R: x --)	
ROLL	(x_u $x_{u-1} \ldots x_0$ u -- $x_{u-1} \ldots x_0$ x_u)	
ROT	(x_1 x_2 x_3 -- x_2 x_3 x_1)	
SWAP	(x_1 x_2 -- x_2 x_1)	
TUCK	(x_1 x_2 -- x_2 x_1 x_2)	

IV.3. Math

-	(n_1/u_1 n_2/u_2 -- n_3u_3)	subtraction
*	(n_1/u_1 n_2/u_2 -- n_3u_s)	multiplication
*/	(n_1 n_2 n_3 -- n_4)	$n_4 = (n_1 * n_2)/n_3$
*/MOD	(n_1 n_2 n_3 -- rem quot)	
/	(n_1 n_2 -- n_3)	division rounds toward zero
/MOD	(n_1 n_2 -- rem quot)	
+	(n_1/u_1 n_2/u_2 -- n_3/u_3)	addition
1-	(n_1/u_1 -- n_2/u_2)	decrement
1+	(n_1/u_1 -- n_2/u_2)	increment
2*	(x_1 -- x_2)	left shift one bit
2/	(x_1 -- x_2)	arithmetic right shift
2+	(n_1/u_1 -- n_2u_2)	alias of CELL+
256*	(n_1/u_1 -- n_2u_2)	left shift eight bits
ABS	(n -- u)	absolute value
AND	(x_1 x_2 -- x_3)	bitwise AND
ARSHIFT	(n_1 u -- n_2)	arithmetic right shift u bits
BASE	(-- c-addr)	returns a byte address! use C@/C!
BINARY	(--)	change numeric base to 2
C>S	(c -- n)	sign-extend byte value to cell
CCOS	(c_1-- c_2)	cosine of binary angle c_1 1.7 signed fixed point result
CHOOSE	(u_1 -- u_2)	random number in range $0 \leq u_2 < u_1$ RANDOM U* SWAP DROP
COIN-FLIP	(-- flag)	random boolean value: RANDOM <0
COS	(c -- n)	cosine of binary angle c 1.15 signed fixed point result.
CSIN	(c_1 -- c_2)	sine of binary angle c_1 1.7 signed fixed point result
DECIMAL	(--)	change numeric base to 10
FM/MOD	(d_1 n_1 -- rem quot)	floored division

IV.3. (cont'd) Math

HEX	(--)	change numeric base to 16
LSHIFT	(x_1 u -- x_2)	left shift by u bits
M*	(n_1 n_2 -- d)	16x16=32 multiplication
MAX	(n_1 n_2 -- n_3)	signed comparison
MIN	(n_1 n_2 -- n_3)	signed comparison
MOD	(n_1 n_2 -- n_3)	n_3 assumes sign of n_1
NEGATE	(n_1 -- n_2)	
RANDOM	(-- u)	random cell-sized integer
RSHIFT	(x_1 u -- x_2)	logical right shift u bits
SEED	(-- a-addr)	address of random seed
SIN	(n_1 -- n_2)	sine of binary angle n_1 1.15 signed fixed point result
SM/REM	(d_1 n_1 -- n_2 n_3)	symmetric division (round toward zero)
U/	(u_1 u_2 -- u_3)	unsigned division
U/MOD	(u_1 u_2 -- rem quot)	unsigned division
U2/	(u_1 -- u_2)	logical right shift one bit
UM*	(u_1 u_2 -- ud)	
UM*/	(ud_1 u_1 u_2 -- ud_2)	
M*/	(d_1 n_1 $+n_2$ -- d_2)	
UM/MOD	(ud u_1 -- urem uquot)	
UMAX	(u_1 u_2 -- u_3)	unsigned comparison
UMIN	(u_1 u_2 -- u_3)	unsigned comparison
UMOD	(u_1 u_2 -- u_3)	
WITHIN	(n_1/u_1 n_2/u_2 n_3/u_3 -- flag)	
XOR	(x_1 x_2 -- x_3)	bitwise XOR

IV.4. Logic

<	(n_1 n_2 -- flag)	signed comparison
<=	(n_1 n_2 -- flag)	signed comparison
<>	(x_1 x_2 -- flag)	test for inequality
=	(x_1 x_2 -- flag)	
>	(n_1 n_2 -- flag)	signed comparison
>=	(n_1 n_2 -- flag)	signed comparison
Ø<	(x -- flag)	
Ø<=	(x -- flag)	
Ø<>	(x -- flag)	
Ø=	(x -- flag)	
Ø>	(x -- flag)	
Ø>=	(x -- flag)	
2SELECT	(d_1 d_2 flag -- d_3)	select d_1 if flag is true, d_2 if flag is false
AND	(x_1 x_2 -- x_3)	
FALSE	(-- Ø)	
INVERT	(x_1 -- x_2)	ones complement
NOT	(x_1 -- x_2)	alias of INVERT
OR	(x_1 x_2 -- x_3)	
SELECT	(x_1 x_2 flag -- x_3)	select x_1 if flag is true, x_2 if flag is false
TRUE	(-- -1)	
U<	(u_1 u_2 -- flag)	unsigned comparison
U<=	(u_1 u_2 -- flag)	unsigned comparison
U>	(u_1 u_2 -- flag)	unsigned comparison
U>=	(u_1 u_2 -- flag)	unsigned comparison
XOR	(x_1 x_2 -- x_3)	

IV.5. Memory

!	(x a-addr --)	store cell-sized value
?	(a-addr --)	print cell-sized value at a-addr
@	(a-addr -- x)	fetch cell-sized value
+!	(n/u a-addr --)	add to cell-sized value at a-addr
2!	(x_1 x_2 a-addr --)	store double-cell value
2@	(a-addr -- x_1 x_2)	fetch double-cell value
C!	(c c-addr --)	store byte
C!+	(c-addr$_1$ c -- c-addr$_2$)	store byte and increment address note reversed argument order
C?	(c-addr --)	print byte at c-addr
C@	(c-addr -- c)	fetch byte
C@+	(c-addr$_1$ -- c-addr$_2$ c)	fetch byte and increment address
C+!	(c c-addr --)	add to byte at c-addr
CBIC!	(c c-addr --)	clear bits in c in byte at c-addr
CBIS!	(c c-addr --)	set bits in c in byte at c-addr
CBIT@	(c_1 c-addr -- c_2)	test bits in byte at c-addr
CELL+	(a-addr$_1$ -- a-addr$_2$)	add 2 to a-addr$_1$
CELLS	(n_1 -- n_2)	size in bytes of n_1 cells
CXOR!	(c c-addr --)	flip bits in byte at c-addr
MAXMEM	(-- addr)	last address in accessible RAM
MINMEM	(-- addr)	first address in accessible RAM
PAD	(-- c-addr)	address of temporary storage region
U?	(a-addr --)	print unsigned cell-sized value at a-addr)

IV.6. Double-cell values

D-	(d_1/ud_1 d_2/ud_2 -- d_3/ud_3)	double-cell subtraction
D.	(d --)	print double-cell value
D.R	(d n --)	print double-cell value right-aligned
D+	(d_1/ud_1 d_2/ud_2 -- d_3/ud_3)	double-cell addition
D<	(d_1 d_2 -- flag)	signed comparison
D<=	(d_1 d_2 -- flag)	signed comparison
D<>	(d_1 d_2 -- flag)	
D=	(d_1 d_2 -- flag)	
D>	(d_1 d_2 -- flag)	signed comparison
D>=	(d_1 d_2 -- flag)	signed comparison
D>S	(d -- n)	equivalent to DROP
DØ<	(d -- flag)	
DØ<=	(d -- flag)	
DØ<>	(d -- flag)	
DØ=	(d -- flag)	
DØ>	(d -- flag)	
DØ>=	(d -- flag)	
D2*	(d_1 -- d_2)	
D2/	(d_1 -- d_2)	arithmetic right shift
DABS	(d -- ud)	double-cell absolute value
DH.	(ud --)	print double-cell value as 8 hex digits w/leading zeros
DMAX	(d_1 d_2 -- d_3)	signed comparison
DMIN	(d_1 d_2 -- d_3)	signed comparison
DNEGATE	(d_1 -- d_2)	
DU<	(ud_1 ud_2 -- flag)	

IV.6. (cont'd) Double-cell values

DU<=	(ud_1 ud_2 -- flag)	unsigned comparison
DU>	(ud_1 ud_2 -- flag)	unsigned comparison
DU>=	(ud_1 ud_2 -- flag)	unsigned comparison
DU2/	(ud_1 -- ud_2)	logical right shift
M-	(d_1/ud_1 n -- d_2/ud_2)	
M*/	(d_1 n_1 $+n_2$ -- d_2)	
M+	(d_1/ud_1 n -- d_2/ud_2)	
S>D	(n -- d)	sign-extend single cell to double
S>DU	(u -- ud)	zero-extend single cell to double
UM*/	(ud_1 u_1 u_2 -- ud_2)	

IV.7. Output

.	(n --)	print single-cell value
..	(n_1 n_2 --)	print n_1 followed by n_2
."	("ccc(quote)" --)	works in compilation state and interpretation state
.(	("ccc(paren)" --)	
.R	(n_1 n_2 --)	print right-aligned
#	(ud_1 -- ud_2)	convert one digit
#>	(d -- c-addr u)	finish numeric conversion
#S	(ud_1 -- ud_2)	convert remaining digits
<#	(--)	begin numeric conversion
BL	(-- c)	ASCII 32 (space character)
CH.	(c --)	print lsb as 2 hex digits
CR	(--)	print a newline
EMIT	(c --)	print character c
EMIT?	(-- flag)	always returns true
H.	(u --)	print u as 4 hex digits
HLD	(-- c-addr)	byte variable containing the current offset in numeric conversion buffer

II.7. (cont'd) Output

HOLD	(c --)	add char to numeric conversion buffer
HOLDS	(c-addr u --)	add string to numeric conv. buffer
PAGE	(--)	print a form feed character (clears console scrolling region)
PUTC	(c --)	print char without interpreting newlines/control-characters
SERIAL-OUT	(--)	make the serial port the output device
SIGN	(n --)	add sign to numeric conv. buffer
SPACE	(--)	print a space
SPACES	(n --)	print n spaces
TYPE	(c-addr u --)	print string
U.	(u --)	print unsigned cell-sized value
U..	(u_1 u_2 --)	print u_1 followed by u_2
U.R	(u n --)	print unsigned right aligned
UD.	(ud --)	print unsigned double-cell value
VIDEO-OUT	(--)	make the video console the output device

II.8. Input

ACCEPT	(c-addr $+n_1$ -- $+n_2$)	
KEY	(-- c)	get one char from input source; wait if no chars available
KEY?	(-- flag)	true if input source has chars available
KEYBOARD-IN	(--)	make the keyboard the input source
SERIAL-IN	(--)	make the serial port the input source
SOURCE	(-- c-addr u)	
SOURCE-ID	(-- Ø / -1 / n)	Ø if input source is console -1 if input source is a string n>0 if input source is block n

II.9. Control flow

?DO	(n_1/u_1 n_2/u_2 --) (R: -- ctx)	
+LOOP	(n --) (R: ctx_1 -- / ctx_2)	
AGAIN	(--)	unconditional loop back to BEGIN
AHEAD	(--)	
BEGIN	(--)	start a loop
CASE	(--)	begin CASE statement
DO	(limit index --) (R: -- ctx)	start a counted loop
ELSE	(--)	
ENDCASE	(x --)	end a CASE statement
ENDOF	(--)	end an OF clause inside a CASE
EXECUTE	(i*x xt -- j*x)	execute execution token
EXIT	(--)	early return from word (must use UNLOOP as well if in DO-loop)

II.9. (cont'd) Control flow

I	(-- n/u) (R: ctx -- ctx)	innermost DO loop index
I'	(-- n/u) (R: ctx -- ctx)	innermost DO loop limit
IF	(x --)	conditional IF-ELSE-THEN
J	(-- n/u) (R: ctx_1 ctx_2 -- ctx_1 ctx_2)	2nd loop index
J'	(-- n/u) (R: ctx_1 ctx_2 -- ctx_1 ctx_2)	2nd loop limit
K	(-- n/u) (R: cx_1 cx_2 cx_3 -- cx_1 cx_2 cx_3)	3rd loop index
K'	(-- n/u) (R: cx_1 cx_2 cx_3 -- cx_1 cx_2 cx_3)	3rd loop limit
LEAVE	(--) (R: ctx --)	early exit DO loop
LOOP	(--) (R: ctx_1 -- / ctx_2)	end DO loop
OF	(x_1 x_2 -- / x_1)	start OF phrase in CASE statement
RECURSE	(--)	recursively call the word being defined
REPEAT	(--)	end BEGIN-WHILE loop
THEN	(--)	end IF-ELSE
UNLOOP	(--) (R: ctx --)	
UNTIL	(x --)	end BEGIN-UNTIL loop
WHILE	(x --)	test in BEGIN-WHILE-REPEAT loop
XY-DO	(xy --) (R: -- ctx)	2 dimensional DO
XY-LOOP	(--) (R: ctx_1 -- / ctx_2)	2 dimensional LOOP

II.10. Dictionary

,	(x --)	allocate one cell in body of current definition
'	("(spaces)name" -- xt)	get execution token for word
[']	("(spaces)name" --)	compile-time version of '
/CODE	(-- u)	size of contents of code space (bytes)
/NAME	(-- u)	size of contents of name space (bytes)
ALLOT	(n --)	allocate n bytes in body of current definition
C,	(c --)	store byte in body of current definition
EMPTY	(--)	clear entire dictionary (code and name spaces)
FIND	(c-addr -- c-addr Ø / xt 1 / xt -1)	Ø if word not found 1 if word is immediate -1 if word is normal
FIND-NAME	(c-addr u -- nt / Ø)	name token for word, or Ø
FORGET	("(spaces)name" --)	delete word and all words defined after it
FORGET-NAME	(nt --)	delete word with name token nt and all words defined after it
HERE	(-- addr)	next free address in current definition
MARKER	("(spaces)name" --)	create dictionary snapshot
NPØ	(-- addr)	address where name space starts
NPMAX	(-- addr)	upper bound of name space
SHRED	(--)	delete name space (leave code space alone)
UNUSED	(-- u)	number of bytes of free memory
X,	(x --)	store x in code space
XALLOT	(n --)	allocate n bytes in code space

II.11. Defining words

;	(--)	end current definition
:	("(spaces)name" --)	begin newword definition
::	("(spaces)name" --)	begin new compiler word definition (body inline in name space)
:[	runtime: (-- xt)	alias of :NONAME
:NONAME	runtime: (-- xt)	create anonymous definition, leave its execution token
2CONSTANT	(x_1 x_2 "(spaces)name" --)	define double-cell constant
2VALUE	(x_1 x_2 "(spaces)name" --)	define double-cell VALUE
2VARIABLE	("(spaces)name" --)	define double-cell variable initialized to Ø.
ACTION-OF	("(spaces)name" -- xt)	get xt of DEFER word
BUFFER:	(u "(spaces)name" --)	define named buffer of u bytes
C:	("(spaces)name" --)	begin new compile-only word definition
CONSTANT	(x "(spaces)name" --)	define single-cell constant
CREATE	("(spaces)name" --)	define new word with default behavior (pushes address of its body)
CREATE::	("(spaces)name" --)	define new compiler word (body inline in name space) with default behavior
CVARIABLE	("(spaces)name" --)	define byte variable initialized to Ø
DEFER	("(spaces)name" --)	create new DEFER word
DEFER!	(xt_1 -- xt_2)	set xt_1 to execute xt_2
DEFER@	(xt_1 -- xt_2)	get xt executed by DEFER word xt_1
I:	("(spaces)name" --)	begin new immediate word definition (body inline in name space)
IMMEDIATE	(--)	must be used before word's body e.g. : FOO IMMEDIATE ... ;
IS	(xt "(spaces)name" --)	assign xt to DEFER word
TO	(i*x "(spaces)name" --)	assign to VALUE or 2VALUE
VALUE	(x "(spaces)name" --)	create single-cell VALUE
VARIABLE	("(spaces)name" --)	define single-cell variable initialized to Ø

II.12. Compiling words

DOES>	(--)	set runtime behavior of CREATE word
::DOES>	(--)	set runtime behavior of CREATE:: word
[	(--)	enter interpretation state
]	(--)	enter compilation state
];	(xt --)	end a :NONAME definition and execute immediately
<COMPILES	(--)	set custom compilation semantics of word
>BODY	(xt -- c-addr)	undefined if xt represents a non-child word
2LITERAL	(x_1 x_2 --)	compile double-cell literal into current definition
COMPILE-ONLY	(--)	delete word's interpretation semantics; makes current definition "compile-only"
COMPILE,	(xt --)	perform compilation semantics of xt
LITERAL	(x --)	compile single-cell literal into current definition
RUNS>	(--)	end custom compilation semantics of current word indicates beginning of interpretation semantics
POSTPONE	("(spaces)name" --)	compile word's compilation semantics (this one takes a while to wrap your head around)

TODO: explain CREATE ...DOES> ... <COMPILES ... RUNS>...

II.13. Parsing words

[CHAR]	("(spaces)name" --)	compile-time version of CHAR
>IN	(-- c-addr)	returns a byte address, not a cell address as the spec dictates
CHAR	("(spaces)name" -- c)	push ASCII value of next word's first char
EVALUATE	(i*x c-addr u -- j*x)	evaluate Forth code in string
PARSE	(char "ccc(char)" -- c-addr u)	obtain next word from input stream, stopping after next occurrence of delimiter char
PARSE-NAME	("(spaces)name(spaces)" -- c-addr u)	obtain next word from input stream, stopping after next occurrence of delimiter char
QUIT	(--) (R: i*x --)	clear return stack and restart interpreter
REFILL	(-- flag)	fill input buffer from input source
RESTORE-INPUT	(inputsrc -- flag)	restore input source from snapshot on stack
SAVE-INPUT	(-- inputsrc)	save input source snapshot on stack
WORD	(char "(chars)ccc(chars)" -- c-addr)	deprecated, but included for standards compatibility

II.14. Block storage

(Note: Ramforth blocks are 512 bytes (32 columns by 16 rows) instead of 1K)

-->	(--)	load next block in sequence
/BLOCK	(-- u)	alias of CHARS/BLOCK
/EE	(-- u)	size of internal EEPROM storage, in bytes
#BLOCKS	(-- u)	total number of storage blocks (internal EEPROM and external storage)
#EE-BLOCKS	(-- u)	number of internal EEPROM storage blocks
#XM-BLOCKS	(-- u)	number of external storage blocks Ø if no external storage device is connected
BLK@	(-- n)	number of block currently being LOADed Ø if input source is not a block
C/L	(-- u)	chars per line in a block (32)
CHARS/BLOCK	(-- u)	chars per block (512)
COPY-BLOCK	(n_1 n_2 --)	copy contents of block n_1 to n_2
COPY-BLOCKS	(n_1 n_2 u --)	copy u blocks starting at n_1 to consecutive blocks starting at n_2
ED	(--)	reopen most recently edited block in editor
EDIT	(n --)	open block n in the editor
ERASE-BLOCK	(n --)	fill block n with space chars (ASCII 32)
ERASE-BLOCKS	(n u --)	fill u blocks starting at n with space chars
FILL-BLOCK	(n c --)	fill block n with ASCII character c
FILL-BLOCKS	(n u c --)	fill u blocks starting at n with ASCII char c
L/S	(-- n)	number of lines in a block (16)
LD	(--)	LOAD the most recently edited block
LIST	(n --)	print contents of block n
LOAD	(n --)	set input source to block n and interpret
SCR	(-- c-addr)	byte variable containing the number of the block most recently LISTed or EDITed

IV.15. Strings

Forth uses two kinds of strings: address-length pairs (c-addr u) and counted strings (prefixed with a length byte, 255 characters max). There is no support for C-style null-terminated strings, as Forth predates C.

-TRAILING	(c-addr u_1 -- c-addr u_2)	
/STRING	($c\text{-}addr_1$ u_1 n -- $c\text{-}addr_2$ u_2)	
>NUMBER	(ud_1 $c\text{-}addr_1$ u_1 -- ud_2 $c\text{-}addr_2$ u_2)	
BLANK	(c-addr u --)	store u spaces starting at c-addr
C"	("ccc(quote)" --)	create counted string in temp. storage (deprecated by standard)
C>HEX	(c -- cc)	convert c to two ASCII digits in display order (16's digit in LSB, one's digit in MSB)
CMOVE	($c\text{-}addr_1$ $c\text{-}addr_2$ u --)	copy u bytes from $c\text{-}addr_1$ to $c\text{-}addr_2$ in ascending order
CMOVE>	($c\text{-}addr_1$ $c\text{-}addr_2$ u --)	copy u bytes from $c\text{-}addr_1$ to $c\text{-}addr_2$ in descending order
COMPARE	($c\text{-}addr_1$ u_1 $c\text{-}addr_2$ u_2 -- n)	0 if strings are identical -1 if string 1 sorts before string 2 1 otherwise
COUNT	($c\text{-}addr_1$ -- $c\text{-}addr_2$ u)	convert counted string to addr/len
ERASE	(addr u --)	store u null bytes starting at addr
FILL	(c-addr u c --)	store u copies of character c starting at c-addr
MOVE	($addr_1$ $addr_2$ u --)	copy u bytes from $c\text{-}addr_1$ to $c\text{-}addr_2$ without clobbering
S"	("ccc(quote)" --)	compile a string literal evaluates to (c-addr u)
S\"	("ccc(quote(" --)	compile a string literal, interpreting backslashed escape sequences (C-style) evaluates to (c-addr u)
SEARCH	($c\text{-}addr_1$ u_1 $c\text{-}addr_2$ u_2 -- $c\text{-}addr_3$ u_3 flag)	
SLITERAL	($c\text{-}addr_1$ u --)	compile string into current definition
U>HEX	(u -- d)	push 4-byte ASCII representation of u in hexadecimal

IV.16. Binary-coded-decimal (BCD) numbers

Exclusive to Ramforth. b denotes a cell-sized 4-digit BCD value (0000-9999) and bd denotes a double-cell 8-digit BCD value (00000000-99999999).

BCD-	(b_1 b_2 -- b_3)	BCD subtraction
BCD.	(b --)	print BCD number as 4 digits. alias of H.
BCD+	(b_1 b_2 -- b_3)	BCD addition
BCD>ASCII	(b_1 -- d)	alias of U>HEX
BCD>ASCII!	(b c-addr --)	store 4-byte ASCII representation of b to memory at c-addr
BCD10*	(b_1 -- b_2)	left shift 4 bits
BCD10/	(b_1 -- b_2)	right shift 4 bits
BCD100*	(b_1 -- b_2)	left shift 8 bits
BCD100/	(b_1 -- b_2)	right shift 8 bits
BCD100/MOD	(b -- ƀ$_1$ ƀ$_2$)	remainder (ƀ$_1$) and quotient (ƀ$_2$) after division by 100 (ƀ denotes a 2-digit BCD value)
BCD100MOD	(b_1 -- ƀ)	remainder after division by 100
BCD2*	(b_1 -- b_2)	add b_1 to itself giving b_2
DBCD-	(bd_1 bd_2 -- bd_3)	double-cell (8-digit) BCD subtraction
DBCD.	(bd --)	print double-cell BCD number as 8 digits. alias of H. H.
DBCD+	(bd_1 bd_2 -- bd_3)	double-cell BCD addition
MBCD+	(bd_1 b -- bd_2)	add 4-digit b to 8-digit bd_1 giving 8-digit sum bd_2

IV.17. Byte pairs

Exclusive to Ramforth. These words interpret a cell as an ordered pair of bytes.

They are used to represent xy coordinates in graphics and cursor-positioning routines.

^	(c_L c_H -- p)	alias of >LH (matches byte pair literal syntax)
><	(p_1 -- p_2)	swap low and high bytes
>H	(p_1 c -- p_2)	replace high byte of p_1 with c
>L	(p_1 c -- p_2)	replace low byte of p_1 with c
>LH	(c_L c_H -- p)	combine low bytes of c_L and c_H into cell
H+	(p_1 c -- p_2)	add c to high byte of p_1 w/o affecting low byte
HI	(p -- c)	high byte of p
HNEGATE	(p_1 -- p_2)	negate high byte of p_1 w/o affecting low byte
L-H	(p -- n)	low byte minus high byte: $n=p_L-p_H$
L+	(p_1 c -- p_2)	add c to low byte of p_1 w/o affecting high byte
L+H	(p -- n)	sum of low and high bytes: $n=p_L+p_H$
LH-	(p_1 p_2 -- p_3)	bytewise difference: $p_{3L}=p_{1L}-p_{2L}$, $p_{3H}=p_{1H}-p_{2H}$
LH-COS/SIN	(p_1 -- p_2)	$p_{2L}=\cos(p_{1L})$, $p_{2H}=\sin(p_{1H})$
LH.	(p --)	print p in bytepair format followed by a space e.g. 12^34
LH*/	(p_1 p_2 -- p_3)	bytewise product-and-scale: $p_{3L}=(p_{1L}*p_{2L})/256$, $p_{3H}=(p_{1H}*p_{2H})/256$
LH+	(p_1 p_2 -- p_3)	bytewise sum: $p_{3L}=p_{1L}+p_{2L}$, $p_{3H}=p_{1H}+p_{2H}$
LH>	(p -- c_L c_H)	split pair into low and high bytes
LNEGATE	(p_1 -- p_2)	negate low byte of p_1 without affecting high byte
LO	(p -- c)	low byte of p

IV.18. Exceptions

ABORT	(i*x --) (R: j*x --)	perform -1 THROW
ABORT"	("ccc(quote)" --)	pop value from stack, if nonzero, perform -2 THROW and print message ccc
CATCH	(i*x xt -- j*x 0 / i*x n)	
THROW	(k*x n -- k*x / i*x n)	

Exception numbers:

-1	ABORT
-2	ABORT"
-3	data stack overflow
-4	data stack underflow
-5	return stack overflow
-6	return stack underflow
-7	branch out of range
-8	name space overflow
-9	code space overflow
-10	division by zero
-11	dictionary entry too long
-12	argument type mismatch
-13	undefined word
-14	interpreting a compile-only word
-15	invalid FORGET
-16	attempt to use zero-length string as name
-17	pictured numeric output string overflow
-18	parsed string overflow
-19	definition name too long
-20	write to a read-only location
-21	unsupported operation
-22	control structure mismatch
-23	address alignment exception
-24	(not used)
-25	return stack imbalance
-26	compiling an intepret-only word
-27	(not used)
-28	user interrupt
-29, -30	(not used)
-31	>BODY used on non-CREATEd definition
-32	invalid name argument (e.g. using TO on a non-VALUE)
-33, -34	(not used)
-35	invalid block number
-36 thru -40	(not used)
-41	loss of precision
-42 thru -47	(not used)
-48	invalid POSTPONE
-49 thru -55	(not used)
-56	QUIT
-57 thru -79	(not used)

IV.19. Console (Amethyst-specific)

-CURSOR	(--)	hide blinking console cursor
-RVS	(--)	disable reverse video in console output
-WINDOW	(--)	reset scrolling region to full screen bounds
+CURSOR	(--)	show blinking console cursor
+RVS	(--)	enable reverse video in console output
AT-XY	(col row --)	set cursor within scrolling region (for standards compatibility)
CLS	(--)	clear text console and reset scrolling region
CTEXT	(--)	switch to 40x25 multicolor text mode
CURSOR!	(xy --)	set cursor position
CURSOR@	(-- xy)	current cursor position
CURSOR+!	(xy --)	add xy to cursor position
DEFAULT-FONT	(-- font)	address of default ROM font
DELAY	(u --)	pause for u frames (1/60-second intervals)
FAST	(--)	disable SLOW console output
FONT	(-- font)	current font/tilemap
FONT:	("(spaces)name" --)	define a new font
FONT!	(font --)	set font/tilemap
FORM	(-- rows cols)	dimensions of console scroll region (from Gforth)
FRAME FRAMES	(--)	no-op; syntactic sugar to use with DELAY e.g. 1 FRAME DELAY
GLYPH!	(ud_1 ud_2 c --)	set glyph c in current font to the 64-bit (8x8-pixel) pattern in ud_1 ud_2
GLYPH@	(font c -- ud_1 ud_2)	get glyph c from font as 64-bit pattern
HTEXT	(--)	switch to 80x25 mono text mode
MS	(u_1 -- u_2)	convert milliseconds to frames use with DELAY: 500 MS DELAY
SECOND SECONDS	(u_1 -- u_2)	convert seconds to frames use with DELAY: 5 SECOND DELAY
SLOW	(--)	enable "slow" console output; pause and wait for keypress before scrolling screen

IV.19. (cont'd) Console

TBOX	(xy wh --)	draw box using box-drawing characters
TEXT	(--)	switch to 40x25 mono text mode
UNTIL-KEY	(--)	loop back to last BEGIN until any key is pressed
WINDOW	(xy wh --)	set console scrolling region

IV.20. Graphics (Amethyst-specific)

-COLOR	(--)	switch to black-and-white video output (disable colorburst)
+COLOR	(--)	switch to color video output (enable colorburst)
>COLOR	(--) (R: c --)	restore current color from return stack
>COLOR>	(c_1 --) (R: -- c_0)	set current color to c_1 and save previous color c_0 on return stack
CGS	(--)	clear graphics screen and reset current color to white
CLEAR	(c --)	fill graphics screen with color c
COLOR	(-- c)	get current color (graphics and text modes)
COLOR!	(c --)	set current color (graphics and text modes)
COLOR>	(--) (R: -- c)	save current color on return stack (restore with >COLOR)
GMODE	(n --)	switch to full-screen graphics mode n (see Appendix A)
GSPLIT	(n --)	switch to split-screen graphics mode (40x5 text console in lower ¼ of screen)
HLIN	(xy w --)	draw horizontal line starting at xy and extending c pixels to the right
HSV	(c -- c)	convert HSV color value to nearest high-color index
LINE	(xy_1 xy_2 --)	draw line between points xy_1 and xy_2 in current color
LPAL!	(c --)	set palette index used by low-color graphics modes
PLOT	(xy --)	set pixel xy to current color
PSET	(c xy --)	set pixel xy to color c

IV.20. (cont'd) Graphics

RECT	(xy wh --)	draw filled rectangle with dimensions wh and upper left corner at xy
SCREEN	(-- c-addr)	address of the screen buffer
SCREEN!	(c-addr --)	set address of the screen buffer
VLIN	(xy h --)	draw vertical line starting at xy and extending h pixels down
VSYNC	(--)	pause execution until the next vertical sync
XMAX	(-- u)	maximum x coordinate on graphics screen: XRES 1-
XRES	(-- u)	width of the graphics screen in pixels
XY>ADDR	(xy -- c-addr)	convert screen coordinate xy to address of corresponding byte in screen buffer
YMAX	(-- u)	maximum y coordinate on graphics screen: YRES 1-
YRES	(-- u)	height of the graphics screen in pixels

IV.21. Sound (Amethyst-specific)

-SOUND	(--)	disable audio output
+SOUND	(--)	enable audio output
BEEP	(--)	emit simple beep, 133ms long
TONE	(freq duration --)	emit square wave with frequency (0-255) and duration in frames (1-255); pause until tone finishes
TONE!	(freq duration --)	like TONE but continue without waiting for tone to finish

IV.22. EEPROM (internal nonvolatile storage)

EE!	(x e-addr --)	write cell to e-addr in internal EEPROM
EE@	(e-addr -- x)	fetch cell at e-addr in internal EEPROM
EE>BUF:	(e-addr u "(spaces)name" --)	allot buffer and fill with data from EEPROM
EE>RAM	($\text{e-addr}_{\text{from}}$ $\text{c-addr}_{\text{to}}$ u --)	copy u bytes from EEPROM to RAM
EEC!	(c e-addr --)	write byte to internal EEPROM
EEC@	(e-addr -- c)	fetch byte from internal EEPROM
EEFILL	(e-addr u c --)	fill u bytes of EEPROM with byte c starting at e-addr
EETYPE	(e-addr u --)	print string from internal EEPROM
RAM>EE	($\text{c-addr}_{\text{from}}$ $\text{e-addr}_{\text{to}}$ u --)	copy u bytes from RAM to EEPROM

IV.23. Peripherals (Amethyst-specific)

"External storage" refers to an AT25xxx-compatible EEPROM connected to SPI port Ø.

Other storage devices may be supported in the future.

CSPIØ CSPI1 CSPI2 CSPI3	(c_1 -- c_2)	transmit/receive one byte on SPI port
RAM>XM	($\text{c-addr}_{\text{from}}$ $\text{x-addr}_{\text{to}}$ u --)	copy u bytes from RAM to external storage
SPIØ SPI1 SPI2 SPI3	(x_1 -- x_2)	transmit/receive two bytes on SPI port (high byte first)
XM>RAM	($\text{x-addr}_{\text{from}}$ $\text{c-addr}_{\text{to}}$ u --)	copy u bytes from external storage to RAM
XMC!	(c x-addr --)	store byte to x-addr in external storage
XMC@	(x-addr -- c)	fetch byte at x-addr in external storage
XMSTAT	(-- c)	read status register of external storage device
XMSTAT!	(c --)	set status register of external storage device

IV.24. System

ALIGN	(--)	no-op
ALIGNED	(addr -- addr)	no-op
BYE	(--)	exit Ramforth
CHAR+	($c\text{-}addr_1$ -- $c\text{-}addr_2$)	add 1 to $c\text{-}addr_1$
CHARS	(n -- n)	no-op
COLD	(--)	restart Ramforth to its initial cold-boot state
ENVIRONMENT?	(c-addr u -- false / i*x true)	environmental query (see Appendix C)
WARM	(--)	reset interpreter state, but do not clear dictionary

IV.25. Tools

.S	(--)	print contents of data stack as signed values in current base
[.S]	(--)	immediate version of .S
[H.S]	(--)	immediate version of H.S
[U.S]	(--)	immediate version of U.S
DEBUG	(--)	enable debug mode (breakpoint after each instruction)
DESCRIBE	("(spaces)name" --)	display information about word
DUMP	(c-addr u --)	print hexdump of u bytes at c-addr
H.S	(--)	print contents of data stack as hex
RESUME	(--)	exit debug mode and continue execution
SEE	("(spaces)name" --)	decompile word
TRACE	(--)	enable debug trace; print disassembly of each instruction as it's executed
U.S	(--)	print contents of data stack as unsigned values in current base
WORDS	(--)	print list of all recognized words

IV.26. Internal

-1	(-- -1)	the value -1 ($FFFF, all bits set)
-INT	(--)	disable interrupts
-TINY	(--)	exit tiny mode; effectively equivalent to COLD
[COMP']	("(spaces)name" -- ct)	get word's compilation token
+INT	(--)	re-enable interrupts
<RESOLVE	(addr offset-addr --)	resolve backward relative branch
>MARKER	(-- snapshot)	push dictionary snapshot
>RESOLVE	(addr --)	resolve forward relative branch from addr to HERE
Ø	(-- Ø)	the value zero (all bits clear)
COMP'	("(spaces)name" -- ct)	get word's compilation token
CP	(-- a-addr)	address of code space pointer
CPØ	(-- a-addr)	address where code space starts
DP	(-- a-addr)	alias of CP
DPØ	(-- a-addr)	alias of CPØ
MARKER,	(snapshot --)	compile dictionary snaphot into current definition
MARKER>	(snapshot --)	restore dictionary snapshot
NP	(-- addr)	address of the name space pointer
PARSE"	("ccc(quote)" -- c-addr u)	parse quote-delimited string from input stream; building block for S" ." ABORT"
PARSE\"	("ccc(quote)" -- c-addr u)	parse quote-delimited string from input stream and convert escape sequences; building block for S\"
RP!	(a-addr --)	set return stack pointer
RP@	(-- a-addr)	current value of return stack pointer
RPØ	(a-addr --)	bottom of the return stack

IV,26. (cont'd) Internal

SP!	(a-addr --)	set data stack pointer
~~SR@XXXXXXXXXXX(XXXXXXXXXXXX)~~		
SP@	(-- a-addr)	value of data stack pointer
SPØ	(-- a-addr)	bottom of data stack
STATE@	(-- flag)	true if in compilation state
TINY	(--)	enter "tiny" memory model

Appendix A. Graphics modes

This is a list of valid values to be used with GMODE and GSPLIT. Split-screen modes replace the lower fifth of the bitmap display with 40 columns x 5 rows of text. This text console requires 200 bytes.

#	Colors	Width	Height (normal)	Height (split)	Mem req'd (normal)	Mem req'd (split)
0	256	160	100	80	16000	13000
1	256	128	100	80	12800	10440
2	256	80	100	80	8000	6600
3	256	80	50	40	4000	3400
4	256	80	25	20	2000	1800
5	16	160	200	160	16000	13000
6	16	160	100	80	8000	6600
7	16	128	100	80	6400	5320
8	16	80	100	80	4000	3400
9	16	80	50	40	2000	1800
10	4	160	200	160	8000	6600
11	4	160	100	80	4000	3400
12	4	80	100	80	2000	1800
13	B&W	640	200	160	16000	13000
14	B&W	320	200	160	8000	6600
15	B&W	256	200	160	6400	5320
16	B&W	160	200	160	4000	3400
17	B&W	160	100	80	2000	1800

Appendix B. Named colors

In 16-color graphics modes and text modes, the following words can be used to push the appropriate color index, or (the versions suffixed with !) set the current color to that value.

#	(-- c)	(--)
0	BLACK	BLACK!
1	D.GREEN	D.GREEN!
2	D.BLUE	D.BLUE!
3	BLUE	BLUE
4	RED	RED
5*	GRAY	GRAY!
6	PURPLE	PURPLE!
7	L.BLUE	L.BLUE!
8	BROWN	BROWN!
9	GREEN	GREEN
10*	GREY	GREY!
11	AQUA	AQUA!
12	ORANGE	ORANGE!
13	YELLOW	YELLOW!
14	PINK	PINK!
15	WHITE	WHITE!

* Colors 5 (GRAY) and 10 (GREY) are visually identical, due to the way the video hardware works.

The following values can also be used in text mode, to select a background color other than black. i.e. BLUE/RED selects blue text on a red background.

16	BLUE/D.GREEN	BLUE/D.GREEN!
17	AQUA/D.GREEN	AQUA/D.GREEN!
18	YELLOW/D.GREEN	YELLOW/D.GREEN!
19	WHITE/D.GREEN	WHITE/D.GREEN!
20	BLUE/D.BLUE	BLUE/D.BLUE!
21	AQUA/D.BLUE	AQUA/D.BLUE!
22	PINK/D.BLUE	PINK/D.BLUE!
23	WHITE/D.BLUE	WHITE/D.BLUE!
24	BLUE/RED	BLUE/RED!
25	YELLOW/RED	YELLOW/RED!
26	PINK/RED	PINK/RED!
27	WHITE/RED	WHITE/RED!
28	AQUA/BROWN	AQUA/BROWN!
29	YELLOW/BROWN	YELLOW/BROWN!
30	PINK/BROWN	PINK/BROWN!
31	WHITE/BROWN	WHITE/BROWN!
32	WHITE/BLUE	WHITE/BLUE!
33	WHITE/GRAY	WHITE/GRAY!
34	WHITE/PURPLE	WHITE/PURPLE!
35	WHITE/GREEN	WHITE/GREEN!
36	WHITE/GREY	WHITE/GREY!
37	WHITE/ORANGE	WHITE/ORANGE!