ColorForth: A powerfull programming language for the COCO-II [1]

G.-E. April Ecole Polytechnique de Montréal P.O. Box 6079 Station A Montreal H3C-3A7 Canada (514) 340-4877

The Radio Shack Color-Computer may be easily modified to accept a Rom-resident version of Forth, especially developped for that machine.

The software has been so arranged as to facilitate the direct transfer of debugged programs into ROM for operation on a stand-alone dedicated board for turn-key operation. Two companion boards were also designed, one being an I/O extender which plugs into the cartridge port of the computer, the other a full featured SBC which shares the same memory and I/O map as the computer (+I/O extender). This allows programs to be developped and debugged on the Color Computer, then transferred without modifications to the target board.

-System features.

In spite of the declared intention to make this a dedicated package, with emphasis on real time performances, Color-Forth (as we lovingly call it), was equipped with a full complement of mathematical and scientific functions as well as string handlers. It also includes some novel structures as well as minor modifications to some classical features of Forth.

-Modified Forth features.

Because of the introduction of complex arithmetic, for which a "c" prefix seemed appropriate, the Forth words c! c@ and cmove were renamed b! b@ and bmove respectively (for byte-store byte-fetch byte-move).

" user " was modified to relieve the programmer from specifying absolute addresses. Instead, the top of the stack is used to determine the number of bytes to be reserved for the variable so defined.

```
e.g. 3 user X  5 user Y  1 user Z  <ENTER> Ok
     X . Y . Z . <ENTER> 3270 3273 3278 Ok
```

" variable " was similarly modified to facilitate its use with ROM-based programs. Instead of initializing the value from the top of the stack, it will use this to determine the number of bytes to reserve, so it is not necessary to use " allot " when variables of non-standard lengths have to be handled. In this way, if it is desirable to have variables reside outside the dictionnary (should this be in ROM), it is sufficient to add the two definitions below and then compile as usual.

```
2 variable Variable_pointer ( Create a pointer
hex 2000 Variable_pointer ! ( Area for variables to start at $2000
: variable Variable_pointer dup @ constant +! ;
```

Thereafter, try:

```
3 variable X0 1 variable Y0 1 variable Z0 <ENTER> Ok
X0 . Y0 . Z0 . <ENTER> 2000 2003 2004 Ok
```

Nothing else need be changed.

---

[1] COCO-II is a registered trademark of Tandy Corporation.

" compile " and [compile] " have been made "state-smart" so that if they are encountered outside a definition (i.e. in "execute" state),they will simply act as no-operation functions. This produces considerable space savings in some types of functions.

" bmove " ("cmove") has been made "direction-smart" and will now produce correct results, even when ranges overlap. This avoids the necessity for different functions for moves up and down. For instance, the following:

```
1000 1001 10 bmove
```

will effectively move 10 bytes up by one position,rather than producing ten copies of the same byte (which could easily be done with " fill ".

-New features

A number of new or relatively rare features have been implemented in ColorForth. Some inspired from other languages, some new (as far as we know).

" block: " is a defining word which, when used in conjunction with " close " and " ;close " allows the definition of local variables and procedures. The words defined by " block: " behave in all respects as though they were colon definitions. The usage of " block: " is illustrated below:

```
: xxx ; ( Place marker in dictionnary )
block: XYZ
: X1 .... Forth_words ... ;
: X2 .... Forth_words ... ;
code X3 ... Assembler_words .... end-code
subroutine X4 .... Assembler_words ... rts, end-code
close ... Forth_words X1 X2 X1 X3 X1 X2 ... Forth_words ... ;close
Ok
vlist <ENTER>
XYZ                                                    block:
xxx                                                        :
<BREAK> Ok
```

The actual definition of XYZ is found between " close " and " ;close " where the same rules apply as in colon definitions. Notice, however, that none of the definitions performed between " block: " and " close " will be available after " ;close " has been successfully compiled. This prevents the dictionnary from becoming cluttered with the names of local variables and procedures.

-The " case " structure.

This is just a convenient way to implement decision trees.

The general format is as follows:

```
.... case ...  of ... Forth_words ... endof
          ...  of ... Forth_words ... endof
          ...  of ... Forth_words ... endof
          endcase ...
```

The word " case " simply marks the start of the structure. The word " of " compares the top two entries on the stack and, if they are equal, drops both and executes the sequence between " of " and " endof ", after which execution resumes at the word following " endcase ". If they are different, the top one only is dropped, and execution continues after " endof ". If execution reaches " endcase ", i.e. if none of the " of "

clauses were satisfied, then the top stack entry is dropped, and execution proceeds past " endcase ". This structure could of course be replaced by a series of " if ... else ... then " statements, but these can get pretty confusing when the number of choices becomes large.

-The " case: " defining word.

This is a defining word that creates words that are decision trees. For cases that are to test all or most values of a variable (from 0 up), this produces more compact and faster code than the " case " structure. The general format is as follows:

```
case: XYZ word0 word1 word2 ..... wordn ;case
```

When " XYZ " is executed, it will take the top stack item and use it to determine which of words0 to wordn should run. Specifically, if the top stack item is 0 or negative, word0 (only) xill be executed. If it is 1, word1 (only) etc. . If it is equal to or greater than n, then wordn will be executed.

-The " defer: " defining word.

" defer: " is very similar in structure to " block: " except for the fact that words defined using it are immediately available for compiling into other words, and the words defined after it remain available. If the word defined by " defer: " is executed, however, nothing happens. The actual action of the word may be defined at a later time, and all previous references will automatically be updated. Further, it is also possible to change the definition many times, without recompiling the program. This allows top-down programming as well as some rather fancy automata. The general format is as follows.:

```
defer: XYZ
: X1 ... XYZ ... ... ;
....
....
define XYZ ... Forth_words ... ;define
```

The actual definition of XYZ is found between " define " and " ;define " with the exact same syntax as a colon definition. However, contrary to colon definitions, all past references to XYZ are updated. To change the definition, it is necessary to first " undefine " the word which may then be " define "d again.

e.g. undefine XYZ define XYZ ... ... ;define

-The assembler.

ColorForth includes a full featured assembler, and two defining words that produce assembler language sequences. The first, " code ", is the simplest but it is a state-smart word on which we shall comment later. Suffice it to say for now that when using " code " as a defining word, the programmer is responsible for preserving the pseudo-program-counter and returning to " next " at the end. " subroutine " on the other hand, uses a high level handler which saves and restores this register. The words defined by " subroutine " must simply end with " rts, ". There is a slight time penalty to these words when they are called by high level definitions, but they have the advantage that they may, without penalty, be called by other assembler language sequences.

```
e.g. subroutine XYZ .. .. .. .. rts, end-code
     : xxx .. .. XYZ .. .. ;
     code .. .. call XYZ .. .. next jmp, end-code
```

As mentionned earlier, code is a state-smart word. When encountered in the execute state, it is a defining word, as seen above. However, when it is encountered inside a definition, it is a kind of switch which allows the definition to proceed in assembler language. The word " Forth " allow the programmer to revert back to high level when the time critical section is finished. The same pair also allows a word defined in assembler to temporarily go to high level, then revert back to assembler.

```
e.g. : XYZ .. .. Forth_words .. .. code .. .. Assembler_words ...
     ... Forth .. .. Forth_words ... ... ;
     code xxx .. .. Assembler_words .. .. Forth XYZ .. Forth_words
     ... code .. .. Assembler_words ... next jmp, end-code
```

-Other features.

Color forth uses two data stacks, one for 16 and 32 bit integers, the other for floating point and complex numbers. A full complement of floating point operators (f+ f- f* f/ fdup fdrop fswap etc.) is included as well as trigonometric and exponential functions (sin cos atn log ln exp etc.). Further most of these operators have a complex equivalent (c+ c- c* c/ cswap etc.) and even exponential and logarithmic functions have a complex equivalent (clog cln cexp ... etc.).

Performances.

The CPU in the ColorComputer is a 6809, which is very well suited to an implementation of Forth. For exemple, it allows the inner interpreter " next " to be implemented in only four instructions and twenty cycles, which, since most Color-Computers will run at 1.8 MHz., adds up to an overhead of little more than 11 micro-seconds.

Typical execution times were measured for the words below (with computer in " fast " mode. These compare favorably with universally accepted standard personnal computers (which shall remain nameless). Further, it should be noted that scientific functions which are seldom used in time-critical situations were not optimized for speed, but compactness and carry 2.5 more accurate digits than most personnal computers.

```
dec ( decimal )
: xx 30000 0 do loop ; ( 1.5 seconds i.e. 50 micro-secs. per loop
: yy 1000 0 do e sin fdrop loop ; ( 25 seconds i.e. 25 milli-secs each
: zz 1000 0 do pi sqr fdrop loop ; ( 5 seconds i.e. 5 milli-secs each
```

Conclusions.

When equipped with our in-house version of Forth, the Color-Computer becomes a powerfull development tool that allows the debugging of dedicated controllers which may then be run unmodified on a target dedicated board, making it much easier to produce powerfull smart devices.