MacADAM DOCUMENTATION:

CONTENTS:

I. SETTING UP TO WORK

To load MacADAM, simply turn ADAM on via the rear of the printer, insert MacADAM into Data/Disk Drive 1 and pull the COMPUTER RESET button located on the top of the Memory Console.

MacADAM will now automatically execute.

The program will have fully loaded when the main menu is displayed, giving 5 options.

The background and text coloring can now be altered by using the cursor keys.

II. USE

II.0. MAIN MENU

The main menu gives you 5 options:

1. Menu of environable parameters
2. Menu for Memory Initialization
3. Menu for Method Printing
4. Menu for I/O Control
5. Menu of Assembling

II.1. THE ENVIRONMENT

This menu allows you to choose in what configuration you will be working. The choices include the selection of the R/W units for memory, the printer, the choice of paper and the number of lines printed per page. The menu also allows you to have 'reading' at the beginning of your program lists.

To alter an option, simply position the cursor to the appropriate line by using the UP and DOWN cursor keys and then type in your option. Hitting ESCAPE takes you back to the main menu. When you introduce a heading, you can use the symbols of the keyboard as well as the LEFT and RIGHT cursor keys.

II.2. INITIALIZATION MENU

This menu simply asks you if you wish to clear the work buffer containing your programs. It's task is to avoid an 'accidental' erasure of your workspace.

II.3. EDITING MENU

The editing menu's function is to allow you to work on a full screen. The screen is a window in your text which will allow you to alter or add to this text. Each sentence is displayed on 2 lines which enable you to write sentences of up to 48 characters long. Each sentence is seperated by the following of a RETURN symbol.

The touch of a key will induce the insertion of a character where the cursor is placed.

The movement of the cursor is made with the help of the cursor keys which are situated on the right of the keyboard.

The ESCAPE key returns you to the Main Menu.

SmartKEYS I & II are connected and can be used in the following way:
SmartKEY I allows you to find text. When SmartKEY I is used, SmartKEY II will position itself at the beginning of the character sequence introduced by SmartKEY I. If SmartKEY I has not been used then SmartKEY II will be stored at the end of a card index.

SmartKEY III is connected to the MOVE command key. A press of SmartKEY III will mark the character where the cursor is currently at. A second touch will erase this mark.

The MOVE key will then move text from in-between the first marked character and the second marked character to the position after any third marked character. If 3 marks have not been made, then nothing will happen. And if there are more then 3 marked characters then the text transfer will only occur using the first 3 marked characters.

The PRINT command key will give you a hard-copy of your text.

The DELETE command key will erase the remaining line on which the cursor is positioned.

The CLEAR command key will erase all workspace after the position of the cursor unless a marked character has been set.

Before returning to the Main Menu, the workspace is analysed and reformed. All blank spaces are erased. All blanks after the labels, code operations and operands are erased and the space code operation is reduced to 4 characters.

The space operand is left unaltered but with the erasure of the blank spaces surrounding it. However, this is only valid for lines which are NOT commentary.

MENU FOR I/O

This menu allows you to communicate with the Menory Module which is at your disposal. The reading and writing are made from the units selected and this menu allows you to load another program after the buffer. If programs musn't be mixed, it is necessary to erase the previous program from the workspace. This menu also allows protection of the original program as well as the produced program.

The later can be carried out in 2 ways:

* NORMAL SAFE-KEEPING *
It's the creation of an indexed file containing the produced file. The produced file is the memory zone included between then minimum and maximum addresses defined by the user's program.

* Safe-keeping of a particular block *
In this case, safe keeping takes place on the block that will be designated by you. The program doesn't check the logic of the inputted number that you introduce and therefore will try to safe-keep. This number must be an integer from OH to FFFFH. The program also doesn't check whether your list is too long or too short (see Particular to the ADAM System). The program will ask you to input the name of the file to be loaded or saved. You can use all keyboard characters as well as the BACKSPACE key to erase any wrongly inputed characters. RETURN marks the end of your introduction. ESCAPE provokes the dismissal of the safe-keeping or of the loading.

II.5. ASSEMBLY MENU

This program allows you to release the assembly and release some lists. Note that with the exception of the Simple List, the other lists require the program to be accurate.

A summary of the options on this level are explained below.

1. Assembling

With this choice only Error messages will be printed (if any).

2. Assembling with complete list

Acts as the previous but supplies a complete list of instructions including those generated by macro-instruction. The list will only appear if there wasn't any errors in the first two steps. It's target is to validate the created macro-instruction.

The same as 2, but the macro-instructions are not displayed. The list will only appear if there wasn't any errors in the first two steps.

4. Simple List

This shows the complete list of the program.

5. Hexadecimal Dump

Supplies the Hexadecimal codes of the produced list. The list will only appear if there isn't any errors in the program.

6. Table of symbols

Will supply the table of symbols used in the program. The list will only appear if there hasn't been any errors in the program.

7. Print

The background color of the menu changes when this option is selected. The way your lists are being printed depends on your choice in the Environment Menu.

In principal, if you have chosen some running lists it isn't possible to interupt the work. In other cases, you will have to depress a key for the work to carry on.

At this level, a press of the ESCAPE key will suspend the works.

III. THE CONVENTIONS

III.1. Instruction Format

The text lines have a maximum of 48 characters. We will notice two types of formats:

* The commentaries *
They begin with * in column 1 and can be followed by 47 characters. They are not analysed and are intended for illustration to the program.

* The real instructions *
They are made of 3 physic fields:

1) The Field Label

Formed by 6 characters, a label is a name permitting to refer to the memory zone where the instructions will be gathered. The first character must be a

letter of ASCII 3/8=41H.
The next five are codes ASCII 3/8=30H.
Special Cases:
The labels used to refer to the macro-instruction must have a maximum of 4 characters because of intervention in the field code operation.

2) The Field Zone Code Operation

Formed by 4 characters. It is intended to receive the Z80 operation codes, the pseudo-instructions and also the macro-instructions.

3) The Last Field; With 2 Functions

First of all, it is necessary to find the operand connected with the mnemonic, then seperated by at least one space an optional commentary area and not included by the analyst.

III.2. Different Operands

1) The operands connected with the Z80 assembly language.
   They are made to specify a mnemonic (ex: LD A,H).

2) The operands in the form of an Octet (Oct).
   They intervene in some Z80 instructions (LD A,3) and in the pseudo-instruct.

3) The operands in the shape of words with 2 octets (word/address).
   They intervene in some Z80 instructions (LD DE,3) and in several pseudo-instructions.

4) The operands linked with a call of a macro-instruction.
   It's a chain of characters without any space or comma. They are seperated by commas and their description is ended by a space or RETURN. It is the same for the operand character of the pseudo-instruction-MOVE and CHG

III.3. Definition of Command

The operands or 'octet': they obey to the same rules as the operands 'mot', but only a small proportion of the octet is kept. The operands 'mot': they are arithemtical phrases without brackets and only containing + or -.
The following can be used:

- some decimal values: written as such
  some hexadecimal values: starting with #
- some binary values: starting with !
- the character F: the associated value is the value of the address where the instruction implants itself.
- the characters: they are surrounded by quotes (' ').
- the 'mots': their value is the one affected to the 'mot' and has been defined by two of the following ways:

1) If they are instruction labels, their value is of the address where the instruction will implant itself.

2) If they are pseudo-instruction labels 'EQU', their value will be of the

pseudo-instruction operands.

- the variables %0 and %1: they are replaced by the last given value. Note: Their initialization is probematical.
- the macro-instruction's operands: the system runs the parameters transition making sure that their number is correct. It's only at the assembling point that an error will be noticed. The error will be at this level and not at the level of the call.
- Definition of the operand's value 'mots'.

NOTE: It is important to be able to determine the values of the pseudo-instructions by the first passage of the analyst. The 'mots' must of been explained before their use. DATA and DW escape to this rule. The system works on the 'mots' by carrying out some staggering steps. It NEVER produces any overflow. This way F2345H is explained as 2345H, 'XYZ' as 'YZ'. The absense of an operand is treated as a zero.
LD D, ---> LD D,0 : LD A,"" ---> LD A,0

III.4. Structure Convention

The convention must be made of two parts:

1) The notification of the macro-instructions.

2) The program which is seperated from the macro-instructions by the pseudo-instruction PGM and which finished by END.

IV. THE LANGUAGE

The language is made up of 3 types of instructions:

1) The pseudo-instructions

2) Z80 Assembly language

3) The call of the macro-instructions as defined by the operator.

The assembling instructions come directly from the machine language where as the pseudo-instruction serves to control the development of the assembling.

IV.1. False Instructions

ASC: syntax LABEL ASC 'TEXT'
Controls the ascii codes of the text put between the next quotes.

CHG: syntax CHG CHARACTER, CHAIN CHARACTER
Permits to modify the variable the &6 by traduction by replacing the first character of the chain. If the result exceeds 6 characters, the chain is abridged. EX: if &6 = ABCABC

```
CHG A,E ---> &6 = EBCEBC : CHG A,C ---> &6 = CBCCBC
CHG F,A ---> &6 = ABCABC : CHG A,EF  -> &6 = EFBCEF
```

DATA: syntax LABEL DATA OCT1, OC2 etc
produces the display of each indicated number on an octet.

DO: syntax DO n
Assemble n times the text situated after the line DC and before the pseudo-instruction ENDO. It is possible to overlap some loops although the analyst will not make any rejection.
Each use of DO provokes the initialization of a specific variable which will decrement itself after it comes across ENDO and provokes an analysis after the last run across DO.

```
EX: DO 8           INC HL
    INC HL         DEC D
    DO 3   GENERE  DEC D
    DEC D          DEC D
    ENDO
```

The first DO isn't taken into consideration.

DS: syntax DS OCT1, OCT2
Permits to generate OCT1 octets initialized to the value OCT2. OCT1 must be completed as early as it's meeting.

DW: syntax LABEL DW MOT
Generates the representation of the number onto 2 octets with the Z80 conventions. Octet of a heavy weight comes after.

EGU: syntax NOM EGU MOT
Permits to allocate a value to a word. MOT must be in the position to be worked out as early as it's first meeting. This pseudo-instruction permits to generate a particular variable to the macro-instruction. The difference with EGU is that a name already affected can be modified by the programmer. At the time of the allocation, the new value will replace the previous one. This possibility, well managed is powerful; but risks may incur as it will be possible to alter the labels. Therefor it is recommended to declare these variables with a name commencing with Z, which will be reserved to them for example.

ENDI: syntax ENDI
Shows the end of the program analysis after an IF.

ENDO: syntax ENDO
Shows the end of a loop (see DO). The program first checks if the variable associated with DO is invalid. If yes, the assembling carries on. If not, the variable is decremented by 1 and analysis will start again at the last position of DO if the variable is still not invalidated. It means that DO 0 is carried out once. This test takes place at the end of the loop.

EQU: syntax NOM EQU MOT
Permits to assign a value to a NOM. MOT should be in a position to be calculated at the first meeting.
This pseudo-instruction permits to manage:
1) Constant - Ex: OPEN EQU FCCO
2) Outer references - Ex:

```
      LIGNE DS 30
      DEBUT EQU LIGNE
      MIL EQU LIGNE+15
      FIN EQU LIGNE+27
```

IF: syntax IF= MOT
Tests to see if the value %1 is equal to MOT. If yes, the text is assembled. If not, the text which follows until the meeting of ENDIF will disappear. MOT must be in a position to be calculated at the first meeting. This instruction permits a conditional assembly. This permits (for example) to opt on certain macro-instructions.
Ex: macro adding the value of the parameter to HL.

```
EXP   MACC 1
      SET 1 &1     EXP 3 genere   INC HL
```

```
      IF< 9                         INC HL
      DO %1                         INC HL
      INC HL
      ENDO          EXP 12 genere   PUSH DE
      OUTM                          LD DE,12
      END1                          DAD HL,DE
      PUSH DE
      LD DE, &1
      ADD HL,DE
      POP DE
      OUTM
```

IF<>: syntax IF<> MOT

Ditto for IF except that the difference is tested.

IF>: syntax IF >=MOT
Ditto for IF except that the superiority or the equality of %1 is tested.
NOTE: -3 is > 4. The 'mots' are considered as positive terms of 16 bits.

MACC: syntax LABEL MACC VAL
VAL must be <=5 & >=0
(see part dedicated to macro-instructions)

MOVE: syntax MOVE VAL1, VAL2, CHAIN CHARACTERS
Permits to initialize the variable &6.
6>= VAL2 >=1 20>= VAL1 >=0
&6 takes the value of the character string, of length VAL2, from character VAL1. The first character is the character 0. If the chain in command has a length inferior to VAL1 + VAL2, some spaces are added.

```
EX:   MOVE 2,2,ABCDEF    &6 = CD
      MOVE 4,1,ABCDEF    &6 = E
      MOVE 3,2,ABCDEF    &6 = DE
```

OUTM: syntax OUTM
Marks the end of a macro-instruction. This provokes a return to the call line with, if the call was a macro-instruction, the parameter's initialization.

PGM: syntax PGM
Marks the beginning of the assembly - it must follow the ,acro-instruction's declaration.
What is preceeding is considered as part of the macro-instruction's declarations. ADRS marks the address at the beginning of the assembly. The variable must be indicated otherwise the system will consider that the assembly begins at 0.

SET0: syntax SET0 MOT
Permits to initialize the %0 variable. MOT must be in a position to be worked out at the first meeting.

SET1: syntax SET1 MOT
Permits to initialize the %1 variable. MOT must be in a position to be worked out at the first meeting.

IV.2. Z80 Assembly Language

For good use of Z80 Assemblt Language, it is advisable to refer to some works giving examples of programming.
The instructions are very simple but are not sufficient by themselves. It's the

complete program structure that will give some results. This will need a method of determination to solve problems of analytical advantages; but there is a certain number of program structures which can be progressively acquired which will help the programming to be as simple as the basic ones.
From this devoted comprehension you will have the satisfaction too see at what speed our programs will go and also to understand the functioning of the printers.

Hereon are the mneumonics used for Z80:

Notation:

NUM designates 00H,08H,10H,18H,20H,28H,30H,38H.
NBIT designates 0,1,2,3,4,5,6,7.
T designates 0,1,2.
OCT designates and octet.
ADR designates an address.
REG designates A,B,C,D,E,H,L.
G designates A,B,C,D,E,H,L,(HL),(IX+DEP),(IY+DEP).
SS designates BC,DE,HL,SP.
QQ designates BC,DE,HL,AF.
PP designates BC,DE,IX,SP.
RR designates BC,DE,IY,SP.
GG designates BC,DE,HL,SP,IX,IY.
CC designates Z,NZ,C,NC,P,M,PO,PE.
(IND) designates the content of IND.

Put X in the system stack means that X is put in the stack and that the scorer of the stack is decremented by 2. The stack works in a decreasing way in the memory.

To recover X in the stack system means that X is taken out of the stack and that the scorer is incremented by 2.

The call for a program means that the inlaid work address following a call is filed in the stack system and that a connection is made at the address of a sub-program.

The return of the sub program is made by the stack to recover the return address.

The flags: They position themselves according to the results from some operations.
Z is the indicator to set to 0 after the operation.
C is the indicator that the capacity has been exceeded after the operation.
P is the indicator of parity after the operation or the overflow.
S is the indicator of sign (SGN) after the operation - if 0 is positive.
H is the indicator that the capacity has been exceeded after the operation to the level of a half octet.

CODE:		OPERATION CARRIED OUT
ADC	HL,SS	HL=HL+SS+CARRY
ADC	A,G	A=A+CARRY
ADC	A,OCT	A=OCT+CARRY
ADD	IY,RR	IY=IY+RR
ADD	IX,PP	IX=IX+PP
ADD	HL,SS	HL=HL+SS
ADD	A,G	A=A+G
ADD	A,OCT	A=A+OCT
AND	G	A=A '& logic' G

```
AND  N            A=A '& logic' OCT
BIT  NBIT,G       put the NBIT of G into Z


CALL ADR          call of program situated in ADR
CALL CC,ADR       call of program situated in ADR if condition CC is realized.


CCF               complement the carry flag.
CP   G            comparison of A and G
CP   OCT          comparison of A and OCT
CPD               compare A and (HL) decrement HL & BC
CPDR              compare A and (HL) decrement HL & BC
                  repeat except if BC=0 or A=(HL)
CPI               compare A and (HL) increment HL and decrement BC
CPIR              compare A and (HL) increment HL and decrement BC
                  repeat except if BC=0 or A=(HL)
CPL               complement A
DAA               adjust the decimal of the accumulator.
DEC  GG           GG=GG-1
DEC  G            G=G-1
DI                forbid interuptions
DJNZ ADR          decrement B and disconnect in ADR is B is different from 0.
                  The address is relative
EI                allows interuptions
EX   AF,AF'       exchange from AF & from AF'
EX   DE,HL        exchange from DE & HL
EX   (SP),IY      exchange from (SP) & from IY
EX   (SP),IX      exchange from (SP) & from IX
EX   (SP),HL      exchange from (SP) & from HL
EXX               exchange of BC,DC,HL, with BC',DC',HL'
HALT              stop of the processor until a RESET or an interuption
IM   T            position of the interuptions
IN   REG,(C)      put the content of port (C) in REG
IN   A,(N)        put the content of port (N) in A
INC  GG           GG=GG+1
INC  G            G=G+1
IND               charge (HL) with the content of port (C). Decrement HL & B
INDR              charge (HL) with the content of port (C). Decrement HL & B
                  repeat except if B=0
INI               charge (HL) with the content of port (C). Increment HL and
                  decrement B
INIR              charge (HL) with the content of port (C). Increment HL and
                  decrement B. Repeat except if B=0
JP   ADR          jump to ADR
JP   (IY)         jump to (IY)
JP   (IX)         jump to (IX)
JP   HL           jump to (HL)
JP   CC,ADR       jump to ADR if the condition CC is realized
JR   ADR          relative jump to ADR
JR   C,N          relative jump to ADR if carry=1
JR   NC,N         relative jump to ADR if carry=0
JR   Z,N          relative jump to ADR if Z=1
JR   NZ,N         relative jump to ADR if Z=0
LD   A,(DE)       A=(DE)
LD   (DE),A       (DE)=A
LD   A,(BC)       A=(BC)
LD   (BC),A       (BC)=A
LD   A,I          A=I
LD   I,A          I=A
LD   A,R          A=R
LD   R,A          R=A
LD   A,(ADR)      A=(ADR)
LD   (ADR),A      (ADR)=A
LD   G,REG        G=REG
LD   REG,G        REG=G
LD   REG,OCT      REG=OCT
LD   GG,(ADR)     GG=(ADR)
LD   GG,ADR       GG=ADR
LD   (ADR),GG     (ADR)=GG
```

```
LD   SP,IY         SP=IY
LD   SP,IX         SP=IX
LD   SP,HL         SP=HL
LDD                put (HL) in (DE), decrement HL,DE,BC
LDDR               put (HL) in (DE), decrement HL,DE,BC, repeat till BC=0
LDI                put (HL) in (DE), increment HL,DE, decrement BC, repeat till
                   BC=0
NEG                A=-A arithmetic (complement of 2)
NOP                'No operation'
OR   G             A=A 'or logic' G
OR   N             A=A 'or logic' OCT
OTDR               put (HL) in port (C), decrement HL,DE,BC, repeat till BC=0
OTIR               put (HL) in port (C), increment HL, decrement BC, repeat
                   till BC=0
OUT  (C),REG       put REG in port (C)
OUT  (N),A         put A in port N
OUTI               put (HL) in port (C), increment HL, decrement BC
OUTD               put (HL) in port (C), decrement HL,DE,BC
POP  GG            recover GG in the stack
PUSH GG            put GG in the stack
RES  NBIT,G        set NBIT of G to zero
RET                return of sub-program
RET  CC            return of sub-program if condition CC is realized
RETI               return of concealable interuption
RETN               return of non-concealable interuption
RL   G             left rotation of bit G with the carry G0=C:C=G7
RAL                left rotation of the accumulator with the carry
RLC  G             left rotation of bit G G0=G7:C=G7
RLCA               left rotation of the accumulator
RLD                left rotation of a half octet between the lower half octet
                   of the accumulator and of the two half octets of HL
RR   G             right rotation of bit G with the carry G7=C:C=G0
RRA                right rotation of the accumulator with the carry
RRC  G             right rotation of bit G, G7=G0:C=G0
RRCA               right rotation of the accumulator
RRD                right rotation of a half octet between the lower half octet
                   of the accumulator and of the two half octets of HL
RST  NUM           call of NUM
SBC  HL,SS         HL=HL-SS-CARRY
SBC  A,G           A=A-G-CARRY
SBC  A,OCT         A=A-OCT-CARRY
SCF                set carry flag to 1
SET  NBIT,G        set NBIT of G to 1
SLA  G             logical left rotation of bit G, G7=G7:C=G0
SRA  G             left rotation of bit G, G7=G7:C=G0
SRL  G             logical right rotation of bit G, G7=0:C=G0
SUB  A,G           A=A-G
SUB  A,OCT         A=A-OCT
XOR  G             A=A 'or logic exclusive' G
XOR  N             A=A 'or logic exclusive' N
```

V. DEFINITION OF MACRO-INSTRUCTION

V.I. Definition of a Macro-Instruction

It's a type of program. A macro-instruction is made up of assembly instructions which are intended to be implanted in the main program; the programmer calls them and so they are not sub-programs. At each call the instruction will be produced. Their target is to avoid recurrences in producing some pseudo-instructions which would help the programmer with the writing of the program. Their strength of these pseudo-instructions will depend on your definitions.

The programmer can saisfy himself by using the macro assembling like a simple

assembling - or even create his own language. This possibility is even greater considering that the tool will help control the development of the assembly. Without having any real border, we can notify several levels of use.

1) A level of management recurrence
2) A level of transition of the parameters of the sub-program
3) A level of advanced autogeneration

The following examples belong to the first 2 levels; the programmers reaching the 3rd level don't need any further advice or getting further documentation on the compilation techniques.

V.2. The Tool

The macro-assembling that you have, disposes of the following possibilities.

V.2.1. Definition of the Parameters

At the declaration of the macro-instruction, we indicate the number of parameters forming the macro-instruction. This number must be between 0 & 5. In the macro-instruction constitution, those parameters will be indicated as &1, &2, &3, &4, &5 and will only be able to define a label, operation code or an operand.

A parameter cannot define 2 fields simultaneously. The rule is that the parameter is replaced in the field where he is showing until he fills the corresponding field. Each call can only define a chain of 6 chracters per parameter.

Ex: If the call of the macro-instruction defines &1 = ABCDEF and if the macro-instruction contains X&1Y, X&1Y, X&1Y : the genere will be : XABCDE, XABC, XABCDEFY.

The label has been mutilated at 6 characters, the operation code at 4 and the operand area preserved whole - the cut only appearing at the 48th character.

There is no restriction for the use of these substitutions. The only rule is that the genere is compatible with the rest of the program and has a meaning. In particular, we can generate a call to a macro-instruction; in return it is important to define one.

V.2.2. Definition if recurrence of the Macro-Instruction

A macro-instruction has the possibility of calling another one, including itself. The parameters of the calling macro-instruction are managed dynamically.

At the return of the call, the parameters will take back their previous value.

```
Ex:
MC1      MACC 1
         INC &1
         OUTM
MC2      MACC 3          ;the call MC2 H,D,SP will generate
         MC1 &1                 INC H
         MC1 &2                 INC D
         MC1 &3                 INC SP
         MC1 &4                 INC H
```

&1 has not been modified from the previous calls. This dynamic management is only possible due to a stack. This stack having a restricted size, the programmer can find himself with an overflow and therefor should find guile. Each call uses 5 octets + 6 octets parameters; the stack presents approximately 8K octets.

Ex: Macro used to push any 4 registers (or less).

```
PUS   MACC 4
      SET1 "&1"    ;%1 takes the ascii value of the content of the
                   ;first parameter
      IF<>         ;test if DE = 0
      PUSH &1
      PUSH &2,&3,&4
      ENDI
      OUTM
```

The call PUS, BC, HL, HL, IX will produce PUSH BC/PUSH HL/PUSH HL/PUSH IX, but will sound (5+6*4)*4 = 116 octets in the stack.

Being less tidy it is preferable to write:

```
PUS   MACC 4
      PUS1 &1
      PUS1 &2
      PUS1 &3
      PUS1 &4
      OUTM
PUS1  MACC1
      SET1 "&1"
      IF<>
      PUSH &1
      ENDI
      OUTM
```

Which only uses 5+24 = 29 octets at the time of a call. To complete this example, observe that the following macro-instructions are even more powerful.

```
PUSA  MACC 4
      PUS1 USH,&1
      PUS1 USH,&2
      PUS1 USH,&3
      PUS1 USH,&4
      OUTM
POP A MACC 4
      PUS1 OP,&1
      PUS1 OP,&2
      PUS1 OP,&3        PUSA AF,HL,, genere
      PUS1 OP,&4        PUSH AF/PUSH HL.
      OUTM              POPA DE,DE,HL, genere
PUS1  MACC 2            POP DE/POP DE/POP HL
      SET1 '&2'
      IF
      P&1 &2
      ENDI
      OUTM
```

V.2.3. The Pseudo Parameter &0

It designates one of the calls parameters'.
The index of the designated parameter is contained in %0. It is advisable to

remember that %0 is a static variable.
Ex: if &1=A,&2=B,&3=C,&4=D

```
      SETO 2
      INC  &0 --> INC &2 ---> INC B
      SETO 4
      INC  &0 --> INC &4 ---> INC D
```

The use of the pseudo parameter is obviously linked to one of the loops.

```
EXPL  MACC 4       is equivalent to EXPL    MACC 4
      SETO 1                                INC &1
      DO 4                                  INC &2
      INC &0                                INC &3
      SETO %0+1                             INC &4
      ENDO                                  OUTM
      OUTM
```

V.2.4. Auxilary Variable Character &6

Like the parameters this variable is a chain of 6 characters (maximum). But contrary to the last ones, it is a static variable.

2 pseudo-instructions allow to proceed on here: MOVE and CHG. Ex: &0 and &6 permit to write the macro-instruction differently; PUSH and POP completing it.

```
OPRG  MACC 5
      MOVE 0,1,&1       OPRG I,B,B,H,H genere
      CHG I,INC         INC B/INC B/INC H/INC H
      CHG D,DEC
      CHG O,POP         OPRG O,BC,BC,AF,HL genere
      CHG U,PUSH        POP BC/POP BC/POP AF/POP HL
      SETO 2
      DO 4              PPRG D,H,H,D,H genere
      SET1 '&0'         DEC H/DEC H/DEC D/DEC H
      IF <>
      &6 &0             PPRG U,AF,HL genere
      SETO &0+1         PUSH AF/PUSH HL
      ENDO
      ENDI
      OUTM
```

V.3. EXAMPLES

V.3.1. Use Of The Variable Statistics %0 %1

These variables especially allow to efficiently use the loops DO and the tests. They can also be used as a coefficient or to manage the labels, but the pseudo-instruction EQU permits to do as much in however offering a larger number of responsibilities. They also allow to recover the parameters of a macro-instruction to test them. In this field we have to be aware of a certain trap.

```
EXEM  MACC 1
      SET0 '&1'
      SET1 &1
```

If the past parameter is A the %0 will take the value corresponding to the ASCII code of 'A' which is 41H.

%1 will take the value corresponding to the address of the label A.

If the past parameter is BCDAA then %0 will take the value corresponding to the ASCII code of 'AA' which is 4141H.
%1 will take the value corresponding to the address of the label BCDAA.

V.3.2. Use Of Recursive Routine

The technique consists in creating a macro-instruction whose name is destined to replace an instruction. This can go from a simple replacement of an instruction which we want to clarify the appelation at the time of creating a specific instruction.

Ex:

```
JPHL  MACC          AHL    MACC          SI   MACC 2
      JP HL                LD A,(HL)          CP &1
      OUTM                 INC HL             JP Z,&2
J=    MACC 1               OUTM               OUTM
      JP Z,&1       XC     MACC 2
      OUTM                 LD HL,(&1)
                           LD (&2) HL
J>=   MACC 1               OUTM
      JP P,&1
      OUTM
```

V.3.3. Use Definition Of A Call Program

The technique consists in passing the parameters of the sub-program as parameters of the macro-instruction which are, in fact, supposed to do the proceeding. This avoids to check how the parameters should be initialized. This is valid for the EOS programs but also for the sub-programs which can be associated with a macro-instruction.
Ex:

```
WRAM  MACC 3        ;call of the print in the VDP
      LD HL,&1
      LD BC,&2
      LD DE,&3
      CALL
      FD1A
      OUTM
CONS  MACC 1              HOME      MACC       CRLF  MACC
      LD A,&1                       CONS 80          CONS C
      CALL                          OUTM             OUTM
      FC39                CUR       MAC 2
      OUTM                          LD D,&1
                                    LD E,&2
                                    CONS 1C
                                    OUTM
```

VI. ERROR MESSAGES

The error messages are connected in the way of how the analysis program works. This analysis is made in 3 fields. Each step is engaged only if the previous one didn't have an error.

1st STEP:

The processor searches for the macro-instruction and the beginning of the program. This can lead to several error messages.

NO PGM: The pseudo-instruction PGM is missing
INVALID NUMBER: You have declared a macro-instruction with more than 5 parameters.
MULTIPLE MACRO: You have declared a macro-instruction with a name already used. Don't forget that the name should have 4 characters. The processor doesn't detect any errors if you use 5 or 6 characters. It will only mutilate the instruction.

2nd STEP:

The processor now knows the macro-instructions; it will try to solve all of the label references.

For this it will have to determine the length of each instruction. At the end, it will see if it has enough spare spaces (blanks) to implant the genere.

INVALID NUMBER PARAMETERS: You don't exceed the number of parameters used. Don't forget that the comma is a seperator.
COMMAND INVALID: The code operation area doesn't tally with a name of the macro-instruction nor with a code operation.
INDEFINED REFERENCE: At this level a compulsory reference is unknown.
INVALID OPERAND: The structure of the operand zone is not valid. This could be a parameter with more than 6 characters or a wrong use of MOVE and CHG.
OUT OF MEMORY: If this happens before the instruction END, it is that there isn't enough room to put the labels in the table. With the instruction END, this corresponds to the fact that there will not be enough room for the genere.
INVALID NUMBER: You used &0 when %0 has a superior value to 6.
STRING INVALID: A chain of characters is used without having been defined.
MULTIPLE LABEL: Label already used.
STACK FULL: Your recurrent calls lead to an overflow of the stack.

3rd STEP:

The macro-assembler is capable of resolving all of the references and can place the program. It will take over step 2 in specifying exactly the genere. This can lead to the finding of new errors.

INVALID REFERENCE: It concerns a reference of a Z80 non-obligatory instruction at the previous level.
INVALID OPERAND: The operand zone is not the one expected.

VII. PARTICULAR TO THE ADAM SYSTEM

Without being able to approach the possibilities of the ADAM system, it isnecessary to know a few points to be able to use the machine language more easily.

1) The Memory Division:

ADAM has an exploitation system (EOS) which enables him to manage different peripherals. This system holds the memory since the address D390H. We can get it by a jump table situated from FC30H to FD50H. The interuptions are situated up to the address 100H. These addresses are at the disposal of the programmer.

They are initialized for return. Note that the 66H (interuptions are not concealed) is released by the VDP every 1/50th of a second.

2) A Few Entry Points:

The keyboard: FD20H. The system sends back the value in the ascii code of the depressed key. There is a waiting of this loop. Normally the Z flag is put to 1, the sequence called is then:

```
LAB   CALL FD20
      JR Z,LAB
```

3) The VDP (video display processor):

It is desirable to know the functioning of the TEXAS INSTRUMENTS TMS9928A processor before using it. It is able to describe the screen memory (16K independent in ADAM: VRAM) in different ways and according to several graphic modes. Different tables exist in VRAM from whom the registers gives the positions. The EOS permits to get to it by the following:

FD20H: Entry of a register, B: Register's number, C: Value to be entered.

FDIAH: Entry in VRAM, HL: Address of the buffer containing the octets, DE: Address in VRAM (16K disponible), BC: Number of octets to be transferred.

FD26H: Entry by filling in VRAM, A: Character to be sent back, HL: Address in VRAM, BC: Number of octets to be transferred.
L EOS: supplies 2 other addresses simplifying the entry of the initialization.

FD26H: Positioning of a table in the VDP, A: Number of the table, 0: Table of sprite attributes, 1: Table of sprites, 2: Screen table of the forms, 3: Drawing table of the forms, 4: Color table of the forms, HL: Position of the table in VRAM.

FD17H: Permits to send the drawing of the ADAM characters in the VDP, HL: Number of the first character to be used, BC: Number of characters to be sent, DEE: Address of VRAM.
Each character uses 8 octets. Even more, the EOS supplies an automatic system of management for the screen window.

FC36H: Permits to initialize this system, B: Number of character columns in the window (<32), C: Number of character rows in the window (<23), D: Number of the first character in the window, HL: Address in VRAM of the screen.

FC39H: Forwarding of a character in the window previously defined (contained in A). Special characters like the basic but with 1C which position the cursor in (D,E).

Initialization sequence:

```
LD B,0        Report registers
LD C,0        Graphic mode 1
CALL FD20
LD B,1
LD C,E0
CALL FD20
LD B,7        Background color
LD C,0
CALL FD20
```

```
LD A,0        Table of sprite attributes
LD HL,1F00
CALL FD29
LD A,1        Table of sprites
LD HL,3800
CALL FD29
LD A,2        Screen table
LD HL,1800
CALL FD29
LD A,3        Table of generators
LD HL,0
CALL FD29
LD A,4        Color table of the generators
LD HL,2000
CALL FD29
LD HL,0       Total character drawings
LD BC,80
LD DE,0
CALL FD27
LD HL,0
LD BC,80      From 80 to FF
LD DE,400     Same drawings as the previous ones
CALL FD27
LD HL,2000    Color of the characters from 0 to 1F
LD A,F0
LD DE,10
CALL FD26
LD HL,2010    Color of the characters from 80 to FF
LD A,0F       Invertion of the previous ones
LD DE,10
CALL FD26
LD BC,1F17    Definition of a screen window
LD DE,0       (here total screen)
LD HL,1800
CALL FC36
```

EX: Write 'OK'.

```
LD A,'O'
CALL FD39

LD A,'K'
CALL FD39
```

Access to the printer: it's made by FC66H, which sends A. It's only necessary to send the codes by ascii. At the return of Z=1 everything is 'OK', otherwise it's better to start again. Be careful; A has been destroyed.
Classical Sequence:

```
TEMPS DS 1,0
PRT   LD (TEMP),A
BCL   LD A,(TEMP)
      CALL FC66
      JNZ BCL
```

Particularities of the indexes:

SmartBASIC permits, with the help of the function BLOAD, to load the code programs. But you must take into consideration that SmartBASIC is in need of some information to be able to load these codes.
Consequently, this means that the index must be contained before the program: 2 octets, specifying the addresses of the insertion of the program. For this it's only necessary to place it at the beginning of the source:

```
PGM  #ADR-5   (Ex: #3000-5)
DATA 1,0,2
DW   #ADR
```

ADR is the address of the insertion and of the execution of the program.
The block zero:
When a 'COMPUTER RESET' happens, ADAM loads the current block zero at the address C800H and releases execution from this address. You can take advantage of this possibility to create some auto-start tape/disks in machine language.

Conception: REGICIEL
Production: Jean-Luc Pronier & Pierre Santoni