Advanced Programming Guide

TABLE OF CONTENTS

ADVANCED PROGRAMMING GUIDE

The catchwords

- chaining between programs
- using shared data and heap
- the assembler interface
- machine level data representation
- better efficiency in programming

apply to this document.

Chapter 1. Introduction

'Advanced programming' as understood in this guide consists of several disciplines, as chaining, sharing data and heap between programs, using assembler modules, and how to get the smallest possible code for a given task.

Chapter 2. Chaining, Shared Data and Heap

To allow you the construction of large program systems, the Modula-2 System for Z80 CP/M features a chaining scheme that allows to share data and the heap between several programs. You can also use the chain scheme without shared data, of course.

Section 1. Chaining Without Shared Data

In this case, you simply use the **Chain** routine from the **Chaining** module to start one program after another one.

A simple example:

```
MODULE FirstOne;

  FROM Chaining IMPORT
       Chain;
    ...
BEGIN (* FirstOne *)
    ...
  Chain( 'SECONDON.COM' );
END FirstOne.
```

```
MODULE SecondOne;

        ...

BEGIN (* SecondOne *)
    ...
END SecondOne.
```

At the end of FirstOne, SecondOne is invoked by the statement CHAIN('SECOND-ON.COM').

To compile link and run this example, use the following command lines:

```
A>MC FIRSTONE <CR>
A>MC SECONDON <CR>
A>ML FIRSTONE <CR>
A>ML SECONDON <CR>
A>FIRSTONE <CR>
```

Section 2. Chaining With Shared Data

Let's go one step beyond: you will learn how to share data between different programs that form a system.

To use this feature, you have to

- provide a **pure data module.** This module consists of a definition module in which you define all the data items you want to be shared between the various program parts, and an empty implementation.
- use the **Chain procedure** as indicated above.
- **link the program's parts** as described in detail later in this document.

First, you will see the above example extended to include shared data.

```
MODULE AssignVariable;

  FROM SimpleShare IMPORT
       SharedCardinal;

  FROM Chaining IMPORT
       CHAIN;

BEGIN (* AssignVariable *)
  SharedCardinal := 10000;
  CHAIN('WRITEVAR.COM');
END AssignVariable.
```

```
MODULE WriteVariable;

  FROM SimpleShare IMPORT
       SharedCardinal;

  FROM InOut IMPORT
       WriteCard;

BEGIN (* WriteVariable *)
  WriteCard(SharedCardinal, 5);
END WriteVariable.
```

As you see, there is a new module involved called SimpleShare. This is the pure data module. It may contain but constants, types, and variables, but no procedures and/or modules. It is a library module consisting of a DEFINITION and an IMPLEMENTATION module. It looks like:

```
DEFINITION MODULE SimpleShare;

  EXPORT QUALIFIED
        SharedCardinal;

  VAR
    SharedCardinal: CARDINAL;

END SimpleShare.
```

```
IMPLEMENTATION MODULE SimpleShare;

END SimpleShare.
```

Compile, link and run is done as follows:

```
A>MC SIMPLESH.DEF <CR>
A>MC SIMPLESH <CR>
A>MC ASSIGNVA <CR>
A>MC WRITEVAR <CR>
A>ML ASSIGNVA/H:5000/S:SIMPLESH <CR>
A>ML WRITEVAR/H:5000/S:SIMPLESH <CR>
A>ASSIGNVA <CR>
```

Here, the linker is told to set the **heap start address** to 5000H by using its **/H** switch. You can tell it the **name of the shared data module** by setting the **/S** switch as indicated in the above example.

NOTE - The heap start address can be set arbitrary; the only thing to watch for is that the heap or the shared data module do not overlay either code or data of one of the programs. If this is case, the linker issues an appropriate error message when linking that program. The messages are:

---- **overlapping code & data segments**
---- **overlapping code & shared segments**
---- **overlapping data & heap segments**

If you set the heap too low, i.e. into the code, you will always get the last of the three error messages listed above.

The lower the heap start address, the larger the heap space that can be used by the program. Keep in mind that the largest program of your system has to fit (including its data area) between the CP/M program start (100H = 256) and the shared data module's start address (see the linker statistics enabled by the /V-switch).

A systematic approach to finding a correct heap start address is

---

- link all component programs as if they were just normal programs. Use the verbose option of the linker (/V).
- for each program part, write down the Data Stop address that can be found in the linker statistics.
- search the largest of these addresses.
- Add at least one to it. You can reserve some space to be able to make additions or corrections also in the largest program of the system without you having to relink the whole system after each change. A typical approach is to set the address to the next page boundary (i.e. instead of 347BH you use 3500H). If and how much space you want to get reserved is up to you.

By following the instructions you just read, you create programs that are able to use one or more variable(s) to communicate.

Normally, each program initializes the heap as it starts up. **To use** also **the heap as a shared data area**, you have to prevent this initialization because it **would destroy the** so called **free list.** This is achieved by means of the **/N** switch of the linker. This switch directs the linker to use an initialization routine that doesn't set the heap.

The free list is used by the heap management to link together all non used heap memory segments. If you would re-initialize it upon start of a program that shares data in the heap, the second program would allocate (by NEW and SYSTEM.ALLOCATE) heap parts that are already used, thus overwriting valid information.

In the next example, the heap is also used as a shared data area.

```
MODULE CreateItem;

  FROM HeapShare IMPORT
       Item,
       SharedItem;

  FROM Chaining IMPORT
       CHAIN;

BEGIN (* CreateItem *)
  NEW( SharedItem );
  WITH SharedItem^ DO
    number := 100;
    alfa := 'x';
  END; (* WITH *)
  CHAIN( 'USEITEM.COM' );
END CreateItem.
```

```
MODULE UseItem;

FROM SimpleShare IMPORT
       Item,
       SharedItem;

  FROM InOut IMPORT
       Write, WriteLn, WriteCard;

BEGIN (* UseItem *)
  WriteCard(SharedItem^.number, 5);
  WriteLn; Write(SharedItem^.alfa);
  WriteLn;
END UseItem.
```

```
DEFINITION MODULE HeapShare;

  EXPORT QUALIFIED
       Item,
       SharedItem;

  TYPE
     Item = POINTER TO RECORD
               number: CARDINAL;
               alfa: CHAR;
            END;

  VAR
    SharedItem: Item;

END HeapShare.
```

```
IMPLEMENTATION MODULE HeapShare;

END HeapShare.
```

To compile, link and run this example, use:

```
A>MC HEAPSHAR.DEF <CR>
A>MC HEAPSHAR <CR>
A>MC CREATEIT <CR>
A>MC USEITEM <CR>
A>ML CREATEIT/H:5000/S:HEAPSHAR <CR>
A>ML USEITEM/H:5000/S:HEAPSHAR/N <CR>
A>CREATEIT <CR>
```

NOTE - One of the most important fact about that shared data and shared heap scheme is that your Modula-2 program source does not contain any dependabilities regarding this special use of the shared module and the shared heap. This ensures that such programs are still portable although in their final compiled and linked form, they include quite a deal of machine dependency.

Chapter 3. Assembler Interfacing

Assembly language interfacing is provided in this system to be able to write really time critical sections of code in assembly language. The code procedure approach used by many implementations was rejected because we think that you should be able to use symbolic assembly language and not criptic hexadecimal or octal codes.

Please restrict use of assembly language as far as possible; our code generator is quite good, although neither perfect nor able to compete with a tricky assembler programmer.

Consider also that a conceptually faster algorithm written in a high level language has a good chance to outperform a slow one coded in fast 'n tricky assembly language.

Section 1. Conventions for Assembler Routines

To integrate assembler modules into your Modula-2 programs, follow these steps:

---

1. Make a normal DEFINITION MODULE describing the interface of your assembler module.
2. Write the assembly language module body. Proceed as described below.
3. Assemble this module with M80 or RMAC, creating a REL file.
4. Write a name translation table according to the specifications given later, if necessary.
5. Convert the REL file with the aid of MR into a MRL file.
6. Link your main program as usual using ML.

WARNING - Although this seems to be very easy, there are several pitfalls to watch for. **If you aren't proficient in assembler, leave writing assembler modules to others.**

Section 2. Names and Name Translation

Because M80 shortens all entries to a maximum of six characters and also uppercases everything, there is an option provided by the REL to MRL format converter MR, that allows translation of such names into arbitrary substitutes. These substitutes are normally qualified identifiers of the form "ModuleIdent.ObjectIdent". The module's name, naturally, consists of the ModuleIdent only.

NOTE - The **data segment** of a module consists only of the entry point **ModuleIdent.ModuleIdent.** All variables are accessed by adding their offset to this base address. **There are no variables that are known "by name".**

NOTE - The Modula-2 linker limits the length of identifiers consisting of module name, a dot, and the object name, to 24 characters overall.

The form of the name translation file is:

```
R2MFile     = Line { CR LF Line } .
Line        = LibReq | RelName whiteChar { whiteChar } MrlName .
LibReq      = '-' FileName .
FileName    = RelName [ CharNum [ CharNum ]] .
RelName     = Char [ CharNum [ CharNum [ CharNum [ CharNum [ CharNum ]]]]] .
whiteChar   = TAB | ' ' .
MrlName     = QualIdent .
QualIdent   = Ident { '.' Ident } .
Ident       = AnyChar { AnyCharNum } .
Char        = 'A' | .. | 'Z' .
AnyChar     = Char | 'a' | .. | 'z' .
Number      = '0' | .. | '9' .
CharNum     = Char | Number .
AnyCharNum  = AnyChar | Number | '$' .
CR          = 15C .
LF          = 12C .
TAB         = 11C .
```

Examples are given later in this section.

Section 3. Segment Usage

The MRL format allows for code and data segments (CSEG, DSEG). COMMON stinks of FORTRAN. No self respecting Modula-2 system would accept such a directive.

Now, some "thou shalt not" statements follow.

Do not use the ORG directive except for the one at the start of each segment (see assembly module template). Because most assemblers default to ORG 0 at the start of any segment, there is normally also no need for that directive.

In the code segment, the loading counter may not be set back by an ORG statement. DS is expanded to DB 0 by the REL to MRL converter.

In the data segment, DB and DW are forbidden, whereas DS can be used. The COM file doesn't contain any data, but only code. This minimizes disk storage space of programs and sacrifices initialized data areas.

Section 4. Variable Access

Please note that the compiler doesn't import names and addresses of variables but rather the start of the data segment and the offset of the respective variables from this start of the data area. These offsets are calculated by aid of the order of the variables in the DEFINITION MODULE, followed by the local variables. This implies, that all local data is allocated **behind** the exported data.

Section 5. Parameter Passing

Parameters are passed on the stack. For VAR parameters, their address is pushed onto the stack, value parameters are copied onto it in full length.

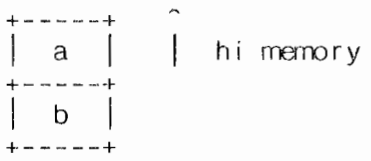
NOTE - The order of the parameters is strictly calculated according to their order in the source text. This implies that the order of the parameters of both

```
PROCEDURE xx(a,b: INTEGER);
```

and

```
PROCEDURE xx(a: INTEGER; b: INTEGER);
```

on the stack before the call is:



This is in contrast to most Pascal implementations.

NOTE - All one byte value parameters are expanded to one word on the stack; the parameter value is located in the low order byte of that word. All other sized parameters are passed in exactly their length, for example 3 bytes or 27 bytes.

Open Array Parameters are passed by a descriptor in the parameter list. This descriptor consists of the address of the parameter and the HIGH-value for the actual procedure call. This looks as follows:

VAR and value parameters are passed the same way. It is the called procedure's responsibility to create a copy of the current value for value parameters. MODLIB

contains the "M-Code" MODLIB.$COPP that expects the descriptor address in HL and the array element size in BC, which copies the actual data and modifies the descriptor accordingly.

Section 6. Function Procedure Result Passing

Before function procedure parameters are loaded onto the stack, space for the function procedure result is allocated on the stack. This, too, is in contrast to most other implementations.

Section 7. Register Usage

The **IX register** is the sacred cow of the high priest of code generation. If you commit sacrilege by altering its value, you will be damned to eternal loops and many other interesting effects.

All other registers (resp. their contents) can be destroyed by your routine(s).

Section 8. External Accesses

You can place freely any external accesses in your assembler modules. The names can be changed on conversion like any other name. There is one point to watch for, however:

WARNING - All identifiers declared to be external (EXTRN ...) have to be referenced at least once. Otherwise, the linker might have difficulties to work on your assembler module.

Section 9. Requesting Library Searches

A module has to request all the modules it depends on. This applies to assembly modules as well as to 'normal' modules. The compiler generates library requests for each module that is used by the currently compiled one.

To be able to do this also in assembly language, MR (the converter) provides your code with such requests. You can insert lines that follow the 'LibReq' production in the name translation file EBNF syntax (see above).

No normal assembly language request ('.REQUEST' in M80) is accepted.

Section 10. Assembly Module Initializations

Initializations of Assembler Modules are called by ML if you write the start label of your initialization code after the END assembler directive. This looks like:

```
END  Init
```

NOTE - Initializations can lie but in the code segment. No absolute addresses may be specified there.

Section 11. Assembly Module Template

An assembler module looks as follows:

```
;
; IMPLEMENTATION MODULE MYMOD;
    .Z80                ; use Z80 operations.
    NAME ('MYMOD')      ;
;
; EXPORT QUALIFIED ....  ;
    PUBLIC   DATA       ; data area (if any).
    PUBLIC   ....       ;
;
```

```
; Module code
        CSEG
        .

;
; Module data
        DSEG
DATA:                       ; has to be translated by MR.
        DS          2       ;    No DB, DW, please.
        .
        .
        END
```

Section 12. Sample Assembly Module

1. Definition Module

```
DEFINITION MODULE Silly;

  EXPORT QUALIFIED eq, EqTest;

  VAR
    eq: BOOLEAN;

  PROCEDURE EqTest( a,b: CHAR );

END Silly.
```

2. Implementation Module

```
      .Z80
      NAME ('SILLY' )           ; file name is SILLY.MAC

      PUBLIC EQTST              ; export procedure
      PUBLIC DATA               ; export data segment

      FALSE EQU  0

      CSEG
EQTST:
      POP      IY               ; return address
      POP      HL               ; "b"
      LD       A,L              ; .. to A Reg
      POP      HL               ; "a"
      CP       L                ; compare them
      LD       A,FALSE          ;
      JR       NZ,L1            ;
      INC      A                ;
L1:   LD       (EQ),A           ; store value in EQ.
      JP       (IY)             ; return.


INI:  RET                       ; just for fun.
```

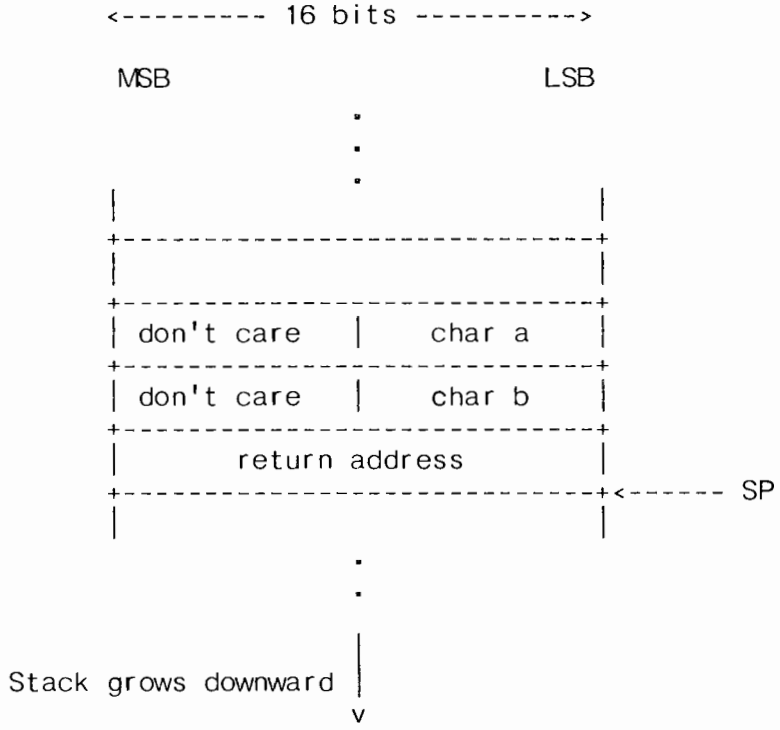
```
        DSEG
DATA:
EQ:
        DS      1

        END     INI
```

3. Name Translation Table

The name translation table looks as follows:

```
SILLY           Silly
EQTST           Silly.EqTest
DATA            Silly.Silly
```

All text in the translation table is left-flushed, i.e. text starts in the first column of each line.



Stack immediately after the call of EqTest.

Section 13. Module Moves

As a second sample, the source of the **Moves** module is presented here. Moves is used to make fast transfers in RAM, i.e. initialization or assignment of arrays.

```
DEFINITION MODULE Moves;

  FROM SYSTEM IMPORT ADDRESS;

  EXPORT QUALIFIED MoveLeft, MoveRight, Fill;

  PROCEDURE MoveLeft(source, destination: ADDRESS; length: CARDINAL);
  PROCEDURE MoveRight(source, destination: ADDRESS; length: CARDINAL);
  PROCEDURE Fill(start: ADDRESS; length: CARDINAL; ch: CHAR);

END Moves.
```

```
;IMPLEMENTATION MODULE Moves;
;
        NAME    ('MOVES')                       ;
;
        .Z80
;
; EXPORT QUALIFIED MVL, MVR, FILL;
;
        PUBLIC  MVL,MVR,FILL                    ;
;
        CSEG                                    ; code to follow.
;-----------------------------------------------------------------------
;
; PROCEDURE MVL(source, destination: ADDRESS; length: CARDINAL);
;
MVL:    POP     HL                              ; ret address
        POP     BC                              ; length
        POP     DE                              ; destination
        EX      (SP),HL                         ; source <---> return address.
        LD      A,B                             ;
        OR      C                               ;
        RET     Z                               ; IF length # 0 THEN
        LDIR                                    ;   MOVE
        RET                                     ; END
;
; END MVL;
;
;-----------------------------------------------------------------------
;
; PROCEDURE MVR(source, destination: ADDRESS; length: CARDINAL);
;
MVR:    POP     HL                              ; ret address
        POP     BC                              ; length
        POP     DE                              ; destination
        EX      (SP),HL                         ; source; push return address back.
        LD      A,B                             ;
        OR      C                               ;
        RET     Z                               ; IF length # 0 THEN
        EX      DE,HL                           ;   dest := dest + length - 1;
        ADD     HL,BC                           ;
        DEC     HL                              ;
        EX      DE,HL                           ;   source := source + length - 1;
        ADD     HL,BC                           ;
        DEC     HL                              ;
        LDDR                                    ;   MOVE
        RET                                     ; END;
;
; END MVR;
;
```

```
;-------------------------------------------------------------------------
;
; PROCEDURE FILL(start: ADDRESS; length: CARDINAL; ch: CHAR);
;
;   or how to use a "typical programming error" to one's advantage.
;
FILL:   POP     HL                      ; ret address
        POP     DE                      ; ch
        POP     BC                      ; length
        EX      (SP),HL                 ; start; push return address back.
        LD      A,B                     ;
        OR      C                       ;
        RET     Z                       ; IF length # 0 THEN
        LD      (HL),E                  ;   Fill first byte
        DEC     BC                      ;
        OR      C                       ; A still "equals" B i.e. BC
        RET     Z                       ; can't be 0 if A # 0.
        LD      D,H                     ;     destination = source + 1
        LD      E,L                     ;
        INC     DE                      ;
        LDIR                            ;     Fill memory
        RET                             ;   END;
;                                         END;
; END FILL;
;
        END                             ;  Moves. No initialization necessary.
```

The file MOVES.R2M takes the following form:

```
MOVES           Moves
MVL             Moves.MoveLeft
MVR             Moves.MoveRight
FILL            Moves.Fill
```

Section 14. Machine Level Data Representation

1. General Considerations

Assembly language programming demands knowledge of the storage layout of the objects to act upon. Therefore, the representation of all standard data types are listed here.

To be able to export variables from an assembler module, one must know how the compiler allocates storage to variables.

In general, variables are allocated just the space necessary to represent them, i.e. a CHAR gets one byte. This is in contrast to parameter passing, where all 1-byte variables are extended to two bytes for efficiency reasons.

2. BOOLEAN

A BOOLEAN variable uses one byte of storage. No packing is done by the compiler.

A formal definition of this type would be:

```
TYPE BOOLEAN = (FALSE, TRUE);
```

Possible values are therefore: ORD(TRUE) = 1, ORD(FALSE) = 0. Only the low order bit is used to determine the BOOLEAN value.

NOTE - On the stack, a BOOLEAN is passed as a 16 bit value. The BOOLEAN value resides in the low order byte.

3. CHAR

A CHAR variable uses one byte of storage.

Possible values: 0C .. 377C.

NOTE - On the stack, a CHAR is passed as a 16 bit value. The CHAR value resides in the low order byte.

4. CARDINAL

CARDINALs are represented as unsigned numbers in the usual Z80 format (low byte in low memory).

Possible values: 00000H .. 0FFFFH, i.e. 0..65535. No overflow detection is done in CARDINAL arithmetic.

5. INTEGER

INTEGERs are represented as signed numbers in the usual two's complement form.

Possible values: 8000H .. 7FFFH, i.e. -32768 up to 32767. No overflow detection is done in INTEGER arithmetic.

6. Enumerations

Enumerations are somewhat tricky. If they fit into a single byte (i.e. if they have 256 elements, at most), their size is one byte. Larger enumerations use two bytes.

NOTE - As all other one byte variables, one byte enumerations are always passed as 2 byte values on the stack.

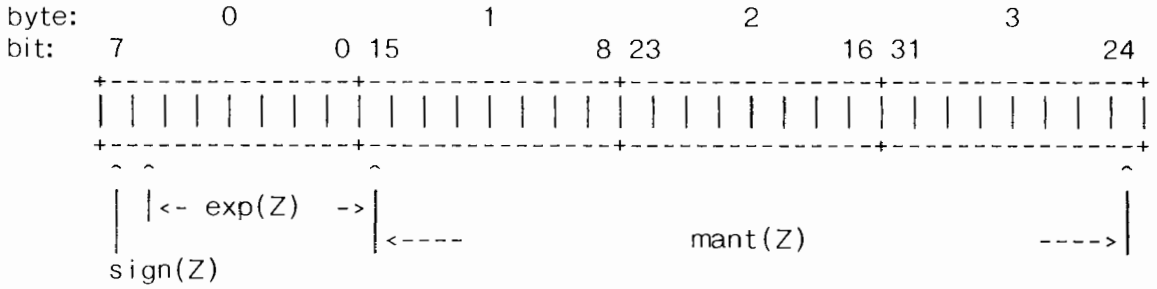
7. Subranges

Subranges have always the same size as their base type. The subrange [0..1] uses still 2 bytes, although it could be represented in a single byte.

Subranges of CHAR etc. are passed as 16-bit values on the stack.

8. REAL

The REAL format selected for use within the Modula-2 System for Z80 CP/M is the so called hidden bit format. This format looks like:



sign(Z) = sign of the mantissa of the number.
exp(Z) = exponent to the base 2 of the number, biased by 80H. 0 value reserved for number 0.

mant(Z) = mantissa of the number with the hi bit always '1' and replaced by the exponent's low bit.
Z = sign(Z) * mant(Z) * $2^{(exp(Z)-80H)}$

In words, this means:

- A hidden bit coded floating point number consists of three parts: the **sign** of the mantissa, the biased **exponent**, and the **mantissa** without the hidden bit. Mantissa and exponent are both coded to the base 2.

- The sign bit is '1' for a negative mantissa, and therefore negative number.

- The exponent may be in the range -127 up to 127. This range is coded as 1..255. The number 0.0 is represented by a zero exponent because there is no way to represent it otherwise. The base (radix) of the exponent is 2, i.e. the mantissa value gets multiplied by $2^{(exponent-80H)}$. The exponent uses 8 bits. The low bit is stored in the hi bit of the second byte of the number.

- The mantissa is 24 bits wide. Because the hi bit is always '1', it is 'hidden', i.e. overwritten by the exponent's low bit. Its value is always in the range 0.5 <= mant(Z) < 1.0.

This format allows positive REAL numbers in the range $0.5 * 2^{-127} = 2^{-128}$ up to nearly $1.0 * 2^{127}$. In scientific notation, this corresponds to about 2.9387E-39 up to 1.70141E38. The resolution is 1 part out of 2^{24}, or about 7.2 decimal digits ($\log(2^{24})$).

Besides the relatively large range for a four byte representation, the hidden bit format also offers the advantage that the significance of the digits sinks monotone, or formulated easier: To compare two hidden bit numbers, you have to care about the sign bit, the rest can be compared by dumb byte-by-byte comparisons. No knowledge of the rest of the representation and also no calculation (i.e. subtraction of the two numbers) is needed!

9. ADDRESS, Pointers and WORD

All of these types are represented as unsigned numbers.

10. BITSET and Sets

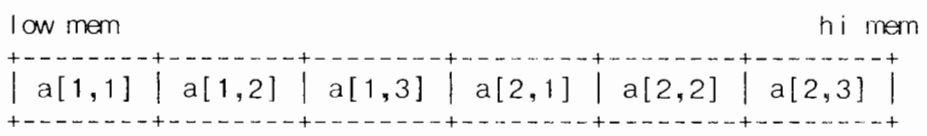
BITSETs and Sets are stored in one WORD, i.e. in 16 bits. each bit corresponds to a set element. The bit numbers 0 up to and including 7 reside in the low order byte, bits 8 up to and including 15 reside in the high order byte. The bit numbers correspond to the order of the bit in each byte, i.e. bit 0 of a set resides in the low order byte bit 0, bit 15 is bit 7 of the high order byte.

11. Arrays

Arrays are represented in row major order, i.e.

```
VAR a: ARRAY [1..2],[1..3] OF CHAR
```

is stored as follows:



Each element uses one byte of storage resulting in SIZE(a) = 6.

12. Records

In Records, each field is allocated exactly its size, i.e. a CHAR typed field uses one byte. If a record variable is only one byte in size, it is expanded to two bytes by the compiler, as mentioned in the beginning of this section.

Variant record variables have their maximum size if they aren't created dynamically by NEW using tagfields. In that case, the size of the particular variant is allocated for such a variable.

If there are closed variants in a type, the maximum size of the closed variant determines the offset of the next field. Closed variants are usually only applied to overlay fields of the same size.

Chapter 4. Programming With Better Efficiency

There are several constructs that ease a programmer's work. But, if you use them in the wrong situation, you can make your code a lot less efficient. Two of the favourite constructions of that kind are the WITH statement and the CASE statement.

Another means to generate more efficient code needs some basic understanding of the compiler's internal work. This is especially necessary for being able to use the strength reduction optimization the compiler performs in expressions to its best.

Section 1. The WITH Statement

The WITH statement basically calculates a base address to access a record once and then saves this address in some local storage. Every access to WITH-referenced items are made by adding the required offset to the once-calculated address.

The fact that this address is calculated once only shows an evident good use of the WITH-statement. Consider the program fragment

```
VAR
  a: ARRAY [0..100, 10..20] OF RECORD
       k,l: CARDINAL;
     END;

 ....

  WITH a[i, j + 27 DIV n] DO
    k := j + 10;
    l := k + j;
  END; (* WITH *)

  a[i, j + 27 DIV n].k := j + 10;
  a[i, j + 27 DIV n].l := a[i, j + 27 DIV n].k + j;
```

The WITH statement does **exactly** the same to the referenced array element, but it saves 3 address calculations.

So, this WITH statement not only pays in code size, it pays also in speed.

Another WITH statement using the same array,

```
WITH a[4, 5] DO
  k := j + 10;
  l := k + j;
END; (* WITH *)
```

is perfectly inefficient. **Why?** It's simple: The compiler not only performs constant expression evaluation, it also evaluates address expressions. So, the address of a[4, 5] can be calculated during compile time. The offsets of k and l, of course, are also constant and therefore, the final addresses of a[4, 5].k and a[4, 5].l can be calculated at compile time. By using the WITH statement, you prevented the compiler from doing this. The resulting code, therefore, is slower and larger than necessary.

Now, let's have a look at procedures and their parameters as WITH statement base addresses.

Consider

```
TYPE Pointer = POINTER TO RECORD
                 x,y: CARDINAL;
               END;


PROCEDURE xx(VAR a: Pointer; b: Pointer);
BEGIN
  WITH a^ DO
    x := .....
    y := ....
  END;
  WITH b^ DO
    x := .....
    y := .....
  END;
END xx;
```

Which one of the two WITH statements is efficient by means of smaller code and faster execution time?

The key is hidden in the difference between VAR and value parameters.

While a VAR parameter is passed by its address (a pointer to it), a value parameter is passed directly as its value.

So, you can easily see that the access to a VAR parameter is always indirect, whereas a value parameter can be accessed directly.

Therefore, we see that the WITH statement around the 'a' VAR parameter removed one indirection by replacing the pointer to the pointer to the record by a pointer to the record. This means that the path to access the value of 'a' is shorter. So, this statement is efficient.

The other WITH statement, the one around b, does not change anything to the access path (there is still the access via a pointer to the record), but adds the code needed to set up the with statement. It might not be much code that is added, but many times a little bit can easily result in 'quite a bit'.

But there are even worse possibilities:

```
TYPE
  IndianGuru = RECORD
                 name: STRING; (* possibly too short .. *)
                 numberOfSupporters: CARDINAL;
               END;

PROCEDURE PrintGuru(guru: IndianGuru);
BEGIN
  WITH guru DO
    WriteString(name); WriteLn;
    WriteCard(numberOfSupporters);
  END; (* WITH *)
END PrintGuru;
```

This sample actually inserted a redirection where there was none before. 'guru' is now accessed via a pointer to it, preventing the compiler from evaluating the addresses of the fields of the record. This is worst case, but you will see that this kind of WITH statements is quite often used for convenience reasons.

The facts brought together:

- WITH statements should be used if they remove indirections by either pointer accesses or array indexing.

- WITH statements for convenience only may be acceptable as long as it doesn't come to program time or space critical applications.

- Be careful with constant addresses; the compiler is intelligent enough to evaluate them in most cases. Do not prevent it from doing its work to its best...

Section 2. The CASE Statement

The CASE statement can be simulated by IF THEN ELSIF THEN etc.- constructs. In fact, that was what the compiler did quite a time.

Now, it generates a table consisting of one entry for each number in the range between the lowest and the highest case label. An extreme (but instructive) example:

```
CASE i OF
  1,2,3: j := 5;

| 10000:    j := 10 + I;
END; (* CASE *)
```

This innocent little case statement generates 10'000 * TSIZE(ADDRESS) = 20'000 bytes of code for the table only ...

So, be prepared: The statement

```
IF (i < 4) AND (i >= 0) THEN
  j := 5;
ELSIF i = 10000 THEN
  j := 10 + I;
```

```
    END;
```

is MUCH smaller than the case statement, although the case statement is almost certainly faster than the IF THEN sequence.

NOTE - One CASE statement cannot have more than 256 labels, overall. This is an implementation restriction. The range covered by these labels is insignificant to this limitation.

Section 3. Constant Expression Evaluation / Strength Reduction

If you had a careful look at the source of ERATOS.MOD, you certainly discovered, that there is a multiplication used instead of an addition in the inner loop. We are not foolish enough to slow down the benchmark by using an unnecessary multiplication where all the others carefully resort to addition.

Please note the order in that expression:

```
    i := i * 2;
```

The compiler recognizes two as a power of two and applies an optimization rule known as **strength reduction.** This rule says that instead of multiplying by N, a power of two, you can use log2(N) shift operations.

So, this multiplication is replaced by a much simpler and faster shift operation.

This rule applies to powers of two only; but in common programming techniques, you use quite often multiplications by powers of two...

The same goes for addition operations. The compiler can optimize only constants that follow variables, not the ones being in front of them.