



**Migrating from Intel ASM-96 and C-96
to
Phyton MCA-96 and MCC-96 Tools**

Application Notes

Migrating from Intel ASM-96 and C-96 to Phyton Tools

Artem Tamazov,
MCC-96 Tech Support

Revision 1.a
April 1999

Contents

1. Intel ASM-96 to Phyton MCA-96 Migration Tips.....	1
1.1. Introduction	1
1.2. Identifiers and reserved keywords	1
1.3. Arithmetic Expressions	1
1.4. Type Checking in Instructions.....	2
1.5. Specifying a Bit Operands	3
1.6. Segmentation	3
1.7. MODULE - Module Definition	4
1.8. EXTRN - Declaration of External Symbols (Identifiers)	4
1.9. CSEG, DSEG etc - Segment selection	4
1.10. EQU, SET - Symbol Definition.....	4
1.11. DCB, DCW etc - Code definition.....	5
1.12. DSB, DSX etc - Storage Reservation	5
1.13. Macro Processing Language.....	6
1.14. What Assembly Startup Does.....	6
2. Intel C-96 to Phyton MCC-96 Migration Tips	7
2.1. Introduction	7
2.2. Initialization at Startup	7
2.3. The "register" Keyword.....	8
2.4. SFR Definition Headers and Other Headers	9
2.5. The "#pragma interrupt" Directive	9
2.6. The "#pragma locate" Directive	9
2.7. Floating-Point Library Issues.....	10
2.8. The "#pragma fixedparams" Directive and GOFAST! Library.....	10
2.9. Tips to Get Your Program Working.....	10
2.10. What C Startup Does	11
2.11. Disabling C Startup	11
3. General Application Tips	13
3.1. Starting Address.....	13
3.2. Stack size.....	13
3.3. Flash Programming Application Note	13
3.4. Understanding Memory Map	18
3.5. Intel RISM and Phyton Tools.....	18

1. Intel ASM-96 to Phyton MCA-96 Migration Tips

1.1. Introduction

The syntax and semantics of MCA-96 directives differ from the Intel's assembler. Hence, you have to convert your existing source files (written for Intel's Assembler) in accordance to MCA-96 rules. The guidelines follow below. If you encounter any difficulties with the conversion, please feel free to contact The Phyton Technical Support.

Note that syntax of machine instructions is the same in both assemblers, therefore you have to convert a relatively small part of source code, including module definitions, data definitions, declarations, and macros. Due to strict type checking implemented by MCS-96, you also have to specifically clarify a strategy for "doubtful" cases.

1.2. Identifiers and reserved keywords

ASM-96 is **not case-sensitive**, and identifiers (symbols) may contain up to 31 characters. Phyton MCA-96 **is case-sensitive**, and identifiers should not exceed 255 characters.

In ASM-96, reserved keywords look like ordinary user-defined identifiers and are case-insensitive. In MCA-96, all the reserved keywords, with the exception of instruction mnemonics, start with a decimal point. All the reserved keywords are case-insensitive just as in ASM-96.

1.3. Arithmetic Expressions

Intel ASM96 supports modulo-64K unsigned arithmetic (i.e., numbers in the form of **unsigned 16-bit** integers). Phyton MCA-96 supports **32-bit unsigned and signed** arithmetics (whether signed arithmetic is used or not depends on current operator; for example, '/' is unsigned division, while '**.IDIV**' is a signed one). Make certain to take this into account when performing assembly-time calculations with numbers greater than 32767.

Example:

```
Const .equ 80h
ldb  AL, #.not Const
```

The Intel tools issue no messages warning about such an error. Our Linker issues the following error message: "FFFFFF7FH out of range".

The reason is very simple. Phyton MCA-96 and the linker (MCLINK) always diagnose overflows while evaluating arithmetic expressions. The arithmetic is a 32-bit. If the result can't fit into the target memory location, Phyton tools issue a diagnostic message. Thus, "Not fit in target" means that the result contains ones (1's) more to the left of on the MSB of target memory location.

In the last example, target memory location has the size of 1 byte. The result of **.NOT 80H** is **0FFFFFF7FH** (32 bits). It can't fit into 1 byte and an error result. The correct syntax is as follows:

```
LDB  AL, #.LOW (.not Const)
or
LDB  AL, #(.not Const) .AND 0FFH
```

This explicitly sets upper 3 bytes of any number to zero. The similar trick may be used for word operands; the **.LWRD** or **.AND 0FFFH** may be used.

Note that Intel ASM-96 silently accepts the original example and the like. Thus, it is always possible to miss an erroneous expression and to hassle with it later.

The most significant differences between the ASM-96 and MCA-96 operators are summarized in the following table:

Intel ASM-96 operator	Hints for migrating to Phyton MCA-96
NUL	Not supported. Use the .BLANK operator instead.
< <= > >= (unsigned 16-bit comparison)	Perform signed 32-bit comparison in MCA-96.
LT LE GT GE (unsigned 16-bit comparison)	.LT, .LE, .GT, .GE perform signed 32-bit comparison. .ULT and .UGT are unsigned versions of .LT and .GT . There are no unsigned versions of .LE and .GE in MCA-96.
SHR (logical shift)	.SHR performs arithmetical shift. For logical one, use the .SHRL operator.

1.4. Type Checking in Instructions

Phyton MCA-96 has *stronger type checking mechanisms* than does Intel ASM-96. The special *operand type* attribute is used to check the correspondence of the instruction code with its operands' sizes. For example, **LDB** instruction solely uses 8-bit operands of **BYTE** operand type

Operand type	Size of operand
BYTE	8 bits
WORD	16 bits
DWORD	32 bits
UNTYPED	unknown

MCA-96 checks whether names are used correctly in instructions. If operand type attribute is not defined for some name, then MCA-96 will consider the name **UNTYPED**. **UNTYPED** operands cannot be used as memory addresses; to do this, special operators should be used to override the operand types. This helps avoid several typical mistakes.

```
ld VAR1,#33      ;the constant is loaded in VAR1
ld VAR2,44      ;Error: no #. Instead of erroneous code
                 ;the Assembler will generate the error message
                 ;because all constants are UNTYPED
ld VAR3,.word 54 ;this is an example of using the .WORD operator
                 ;for overriding of operand type when you need to
                 ;use a number as an address
```

Note that both MCA-96 and the MCLINK check for the correct name alignment if the names have operand type **WORD** or **DWORD**.

To use any name (expression) as an operand in any instruction, you may override the operand type of this name (expression). Note that the **.DCx**, **.DSx**, **.LABELx**, **.EXTRNx** directives automatically assign operand type to the data items defined by them. You may also use the **.BYTE**, **.WORD**, **.DWORD** directives and operators for type (re)declaration.

1.5. Specifying a Bit Operands

Intel ASM-96 supports automatic calculations of base (byte) address and a bit number for the following syntax (Note: ASM-96 allows a comma or a period after 1st operand):

```
RSEG at 30H
bitfields: DSL 1 ;32 bits for semaphores
CSEG
JBS bitfields,28,$
```

Phyton MCA-96 does not agree with this syntax. The following type-safe syntax allows calculating offset and a bit number for 32- and 16-bit "semaphore" that registers automatically:

```
.RSEG REGS,reg ;relocatable register segment
Bitfields .DSD 1 ;semaphores (unsigned 32-bit variable)
MyBit .equ 28
.RSEG CODE,mem ;relocatable code segment
jbs .BYTE (Bitfields+MyBit/8),(MyBit .MOD 8),$ ;.BYTE to suppress warning
;about type conflict:
;"Bitfields" is a 32-bit
;integer
```

1.6. Segmentation

Overlayable register segments are not supported by MCA-96. For further information, see the manual: *Programming with MCA96 → Memory Reservation for VWindows (in the Register File)* and *Programming with MCA96 → Allocation of User's Data in Vertical Windows*.

There are no distinct code, data, and const segments in MCA-96; all are combined in the single `mem` segment type.

Placement of the re-locatable segments does not depend on how absolute segments are placed, thus you have to make sure to prevent the re-locatable segments from overlapping with absolute ones. To do this, use Linker's `-N` option and `.LNKCMD` directive.

1.7. MODULE - Module Definition

Intel ASM-96 syntax: `modulename MODULE [MAIN|STACKSIZE(n)[,MAIN|STACKSIZE(n)]]`

Phyton MCA-96 syntax: `.pmodule|.lmodule|.lmodule2 modulename [,reg|mem]`

Important differences: MCA-96 supports multiple modules in one file. All modules, except the last one, must end with the `.ENDMOD` directive. The `.pmodule` directive defines the program module, `.lmodule` defines the library module, `.lmodule2` defines the low-priority library module.

1.8. EXTRN - Declaration of External Symbols (Identifiers)

Intel ASM-96 syntax: `EXTRN symbolname[:datatype] [,symbolname[:datatype]]...`
 where *datatype* must be `BYTE`, `WORD`, `LONG`, `ENTRY`, `REAL` or `NULL`.

Phyton MCA-96 syntax: `.EXTRNB (mem|reg) name [,name] ... ;for BYTES`
`.EXTRNW (mem|reg) name [,name] ... ;for WORDS`
`.EXTRND (mem|reg) name [,name] ... ;for DWORDS`
`.EXTRNC (mem|reg) name [,name] ... ;for signed BYTES`
`.EXTRNI (mem|reg) name [,name] ... ;for signed WORDS`
`.EXTRNL (mem|reg) name [,name] ... ;for signed DWORDS`
`.EXTRNR (mem|reg) name [,name] ... ;for REALS`
`.EXTRNN name [,name] ... ;for simple numbers`
`.EXTRNF funcname [expr1] [(expr2 [, expr3] ...)] ;for functions`

Important differences: In MCA-96, the segment type of the symbols is set independent of the active segment, in accordance to the `mem` or `reg` attribute. Therefore, you can place these directives any place in your module. The `datatype` attribute is unnecessary. The syntax of `.EXTRNF` is discussed in the documentation; in most cases, omit all after `funcname`.

1.9. CSEG, DSEG etc - Segment selection

Intel ASM-96 syntax: `CSEG|DSEG|RSEG|KSEG|OSEG [REL|AT baseaddress]`

Phyton MCA-96 syntax: `.ASEG|.RSEG segname [,mem|reg]`

Important differences: In MCA-96, each segment has a *symbolic name*. At first, you must specify `mem` or `reg` for each *segname*; in later segment selections this attribute becomes an option. `.ASEG` defines the absolute segments, `.RSEG` defines the relocatable ones. There is no attribute like `baseaddress` in MCA-96; use the `.ORG address` directive instead. There are no distinct code, data, and const segments in MCA-96; all segments are combined in the single `mem` segment type. Overlayable register segments are not supported (see above).

1.10. EQU, SET - Symbol Definition

Intel ASM-96 syntax: `name EQU|SET expression [:BYTE|WORD|LONG|ENTRY|REAL|NULL]`

Phyton MCA-96 syntax: `name .DEFINE expression`
`name .EQU expression`
`name .SET expression`
`name = expression`

Important differences: There are no attributes like `BYTE`, `WORD` etc. in MCA-96. You can use `.BYTE`, `.WORD` and `.DWORD` directives to assign type information to the names defined by `.SET` and `.EQU`. The '=' character is synonym for `.SET`. The `.DEFINE` directive keeps its value in all of the following modules of the source file, while identifier defined by other directives is valid only in the current module.

1.11. DCB, DCW etc - Code definition

Intel ASM-96 syntax: `[label:] DCB expression|string [,expression|string]...`
`[label:] DCW expression [,expression]...`
`[label:] DCL expression [,expression]...`

Phyton MCA-96 syntax: `[name] .DCB expression|string [,expression|string] ...`
`[name] .DCW expression [,expression] ...`
`[name] .DCD expression [,expression] ...`
`[name] .DCC expression|string [,expression|string] ...`
`[name] .DCI expression [,expression] ...`
`[name] .DCL expression [,expression] ...`
`[name] .DCR expression [,expression] ...`

Important differences: The semicolon *must not* be present after *name* in MCA-96. There are signed counterparts of `.DCB/.DCW./DCD`. Note that the synonym for Intel's `DCL` is `.DCD` in MCA-96. However, `.DCL` is OK in most cases since the only difference between them is in the type of information assigned to *name*. MCA-96 also has the `.DCR` directive which allows to enter the floating values (the major difference between the `.DCL/.DCD` is in type information assigned to *name*).

1.12. DSB, DSW etc - Storage Reservation

Intel ASM-96 syntax: `[label:] DSB|DSW|DSL|DSP|DSR expression`

Phyton MCA-96 syntax: `name .DSB expression|string [,expression|string] ...`
`name .DSW expression [,expression] ...`
`name .DSD expression [,expression] ...`
`name .DSC expression|string [,expression|string] ...`
`name .DSI expression [,expression] ...`
`name .DSL expression [,expression] ...`
`name .DSR expression [,expression] ...`

Important differences: The *name* is mandatory in MCA-96; the semicolon *must not* be present after *name*. MCA-96 does not support direct counterpart for Intel's `DSP`, thus you can use `.DSD`. Strings are allowed in the `.DSB` and `.DSC` directives. There are additional signed versions of `.DSB/.DSW./DSD`. Note that synonym for Intel's `DSL` is `.DSD` in MCA-96, but `.DSL` is OK in most cases because the only difference is in type information assigned to *name*.

1.13. Macro Processing Language

There are many differences between Intel ASM-96 and Phyton MCA-96 beyond the scope of this document. Please, read the documentation and contact Phyton Tech Support if you encounter some problems.

1.14. What Assembly Startup Does

Question:

>2. Is there documentation on the elements included in startup? Since this code >will be executing within an Avionics product, the FAA will require that we >have tested and understood all aspects of the code. This means that we must > understand and deliver any code that is placed automatically.

The documentation is not included because we deliver all the sources of our library.

Assembly startup (SOURCE\RESERVED.MCA) does the following:

1. Initializes all the reserved memory locations with 0FFH's
2. Vectors used interrupts to the proper (user-defined) routines.
3. Vectors unused interrupts to the following code:

```
DI
SJMP $
```

4. Initializes CCBs with default values (see documentation on #pragma CCB), if CCBs are not initialized explicitly.
5. Jumps to the **?START** public label (first instruction of your assembly program must have this label). Note that for programs written in C, assembly startup jumps to the C startup.

2. Intel C-96 to Phyton MCC-96 Migration Tips

2.1. Introduction

Phyton MCC-96 greatly differs from Intel C-96 in the area of 80C196-specific support. Other differences are stipulated by the fact that MCC-96 is much more ANSI C-compatible than iC-96. Perhaps you will be compelled to make more changes in your old sources than you have previously expected. That is because old compilers force to use impermissible (from the ANSI point of view) language extensions. Therefore, we are sure that the costs of making necessary changes will be compensated hereafter, since your programs become more appropriate to the language Standard and portable. The last and most important hint: please, read the MCC-96 documentation, especially the following topics:

CHAPTER 2. COMPILING WITH C96
 Linking the C96 Program
 CHAPTER 3. LANGUAGE IMPLEMENTATION
 Memory Models and Pointers
 Conformance to ANSI Standard
 #pragma startup: Execute Function at Startup
 CHAPTER 4. 80C196-SPECIFIC SUPPORT
 #pragma datloc: Location of Static Objects
 #pragma interrupt: Writing Interrupt Service Routines in C
 #pragma CCB/CCB1/CCB2: Setting the Chip Configuration Bytes
 #pragma p5_reg_init/p5_dir_init/p5_mode_init: port 5 mode
 Setting the STACK and HEAP Sizes
 Accessing SFRs from C
 Accessing Internal SFRs via Vertical Windows
 Accessing Absolute Memory Locations
 CHAPTER 7. LIBRARY REFERENCE
 Library Files
 Standard Input, Output and Error Streams
 Include Files Reference

If you encounter any difficulties with the conversion, please, feel free to contact Phyton Technical Support.

2.2. Initialization at Startup

The iC-96 compiler supports initialization of only constant objects at file scope and initialization of only constant and automatic objects at block scope. MCC96 supports all kinds of initialization according to ANSI standard. You may want to remove old explicit initialization code.

2.3. The "register" Keyword

This is about one of Intel's ANSI-incompatible language extensions. iC-96 uses the `register` keyword to place static variables in the Register File. Our compiler has different, ANSI-compatible means:

Syntax:

```
#pragma datloc reg
#pragma datloc mem
#pragma datloc .
```

Discussion:

The compiler can place non-constant static objects in the lower register file or in the external memory.

Specifying the `-r` option will allow compiler to place static objects in the registers. Otherwise, the compiler will use its external memory. It should be noted that constant static objects are always placed in the external memory.

You can use `#pragma datloc` directive to dynamically control the allocation of non-constant static objects directly in the source file:

```
#pragma datloc reg

/* place file level and static variables in the register file */
int var1 = 1;
#pragma datloc mem

/* place file level and static variables in the memory */
int var2 = 2;
#pragma datloc .

/* place file level and static variables in accordance to
the presence of the -r option in the C96 command line */
int var3 = 3;

/* The -r option and #pragma datloc directive also affects the
external definitions of non-constant static objects */
#pragma datloc reg
extern int var4;          /* external register variable */
#pragma datloc mem
extern int var5;          /* external variable in memory */
```

2.4. SFR Definition Headers and Other Headers

One of the most frequently asked questions is:

*>I get an " Error [53], Incompatible storage class" on a file
>(80C196.h) that I believe came from Intel for the 80C196. Can you please
>advise if this file is still used for the 80C196 or does your software
>handle this in a different way. I have attached the 80C196.h file.*

To get things working properly, you have to use SFR's definitions files supplied by the MCC-96 instead of Intel's files. Never use Intel's SFR definitions files and other Intel's headers. As a rule, the headers that are supplied by any compiler are **compiler-specific** and **use excessive vendor-invented language extensions**. Therefore, such headers are nonportable.

To be more concrete, I present the quote below from the Phyton MCC-96 documentation:

The C96 library provides you with a number of include files that contain definitions of Special Function Registers (SFRs) found on all major 80C196 derivatives. These files are listed below:

```
sfrs_bh.h - for 8X9XBH, 8X98, 8X97JF
sfrs_kb.h - for 8XC196KB, 8XC198/194
sfrs_kd.h - for 8XC196KC/KD, 8XL196KD
sfrs_kr.h - for 8XC196KR/JR/KQ/JQ/KT/KS/JT/JS/JV
sfrs_mc.h - for 8XC196MC
sfrs_md.h - for 8XC196MD
sfrs_mh.h - for 8XC196MH
sfrs_ca.h - for 8XC196CA
sfrs_nt.h - for 8XC196NT
sfrs_np.h - for 8XC196NP
sfrs_nu.h - for 8XC196NU
sfrs_cb.h - for 8XC196CB
```

2.5. The "#pragma interrupt" Directive

`#pragma interrupt` has different syntax in MCC-96:

Example for iC-96:

```
#pragma interrupt (UartReceiveISR = 25) /* valid for iC-96 */
```

The same for MCC-96:

```
#pragma interrupt 9 UartReceiveISR /* valid for MCC-96 */
```

2.6. The "#pragma locate" Directive

MCC96 does not support `#pragma locate`. Use the built-in `__BYTE_AT__`, `__WORD_AT__`, `__DWORD_AT__` macros instead:

Example for iC-96:

```
char MyByte;
#pragma locate (MyByte = 0xc002)
```

The same for MCC-96:

```
#define MyByte (__BYTE_AT__(0xc002))
```

2.7. Floating-Point Library Issues

The explicit initialization of the floating-point library after program startup is no longer needed (do not attempt to use `fpinit()`). MCC-96 built-in floating-point library is fully reentrant, so do not attempt to use `fp_save()` and `fp_restore()`. (These functions are not needed and therefore are not implemented).

2.8. The "#pragma fixedparams" Directive and GOFAST! Library

One of the frequently asked questions is:

```
>When I compile my files I get some warning[128] messages for "Unknown  
>#pragma fixedparams" These #pragma are from a module called GOFAST.H. I  
>believe it was used for floating point operations by the previous  
>programmer. Can these #pragma be added somewhere so that  
>the compiler will recognize them? I have attached the GOFAST.H file for  
>your review.
```

You do not need this file anymore. In the past, it was necessary indeed since the GOFAST! floating point library was used in your project instead of the slow Intel's FPAL-96. Now that library is not required nor of need to you, our compiler has it's own built-in support for floating-point numbers, which works 2.7 times faster than FPAL-96.

Now you can write in Standard C using floating-point operations. In other words, that you don't need to `#include` any headers to perform basic operations with floats - multiplying, division, addition, comparison, conversions to integers and back, etc. For trigonometric functions (`cos`, `sin` etc.) the header `<math.h>` is required; this header and appropriate functions are supplied by the MCC-96 Library that you have.

The `#pragma fixedparams` directive is not supported by MCC-96 since it's unnecessary.

2.9. Tips to Get Your Program Working

1. Start your program with "void main(void) {...}" function. Do not attempt to write your own startup code – usually, standard startup is quite suitable. MCC-96 has special means of properly initializing all the special memory locations.
2. Link C runtime library together with your object modules. C runtime library includes all that your program may need – standard startup and initialization, floating-point routines, 32-bit integer arithmetic routines, standard C functions, and so on. There is no need to be concerned with the size of your program since the Linker will include only the code that your program really needs.
3. Specify the addresses and the sizes of RAM and ROM to the Linker. Also, specify the sizes and the locations of the STACK and HEAP segments. (Since HEAP is not commonly used, set its size to 0).
4. The Linker will automatically place your code and constants in ROM, "mem[ory]" objects in RAM, "reg[ister]" objects in the lower register file. Also, it will automatically set up the program start address, interrupt vectors, CCBs, and other special locations (including reserved locations).

2.10. What C Startup Does

Question:

>2. Is there documentation on the elements included in startup? Since this code >will be executing within an Avionics product, the FAA will require that we >have tested and understood all aspects of the code. This means that we must >understand and deliver any code that is placed automatically.

The documentation is not included because we deliver all the sources of our library.

First, assembly startup (SOURCE\RESERVED.MCA) is executed. See [1.14 What Assembly Startup Does](#) for description.

Second, C startup (SOURCE\STARTUP.MCA) is executed. C Startup performs actions that are needed by the C Standard and some of the MCS96-specific actions:

0. Sets up minimal stack size of 40H bytes.
1. Initializes Port5 (only for 196KR/NT; see `#pragma p5_XXX_init`).
2. Initializes the processor's stack pointer (`SP`) and sets up several tables needed for steps 3,4,5.
3. Initializes 'initialized' static variables
4. Clears 'uninitialized' static variables.
5. Calls functions mentioned in the `#pragma startup` directives. Several library functions may be called at this stage; which ones – depends on the user's application. For example, if free storage (heap) is used in your program, `??HEAP_INIT` is called which it initializes memory pool for heap (the `HEAP` segment).
6. Next, C startup calls `main()`.
7. If `main()` exits, C startup loops forever.

2.11. Disabling C Startup

Question:

>Is there a way to disable Phyton from entering the startup code and jump >directly to main()?

Yes. You have to add the following **assembly module** to your C program:

```

////////////////////////////////////
;          C STARTUP MODULE          ;
;          'TRUNCATED'              ;
;          ALL MCS-96 CHIPS AND MEMORY MODELS ;
;          (C) 1996, PHYTON          ;
////////////////////////////////////

.LMODULE ??EMPTY_STARTUP

.include 'tmpreg.inc'

.rseg CODE_??CLIB          ,mem
.rseg CONST_??CLIB        ,mem
.rseg MDATA_??CLIB        ,mem
.rseg ??INIT_FUNC_PTR_TABLE ,mem
.rseg HEAP                 ,mem
.aseg ??PTSVECTORS        ,mem
.rseg RDATA                ,reg
.rseg IRDATA               ,reg
.rseg URDATA               ,mem
.rseg IURDATA              ,mem
.rseg IRDATA_INI           ,mem
.rseg IURDATA_INI         ,mem
.rseg ??INIT_DATA_SEGMENT_ADDRESSES ,mem
.rseg ??DATA_SEGMENT_ADDRESSES ,mem
;

```

```

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Edit the following .lnkcmd directives to move segments
; into another address areas, if needed.
;
; * WARNING * Be sure you are really know what you are doing!
;
.lnkcmd '-S RAM(MDATA_??CLIB)'           ;shall be placed in RAM
.lnkcmd '-S RAM(HEAP)'                   ;shall be placed in RAM
.lnkcmd '-S default_reg(RDATA)'         ;shall be placed in registers
.lnkcmd '-S default_reg(IRDATA)'        ;shall be placed in registers
.lnkcmd '-S UPPER_REGFILE(URDATA)'      ;shall be placed in the upper rfile
.lnkcmd '-S UPPER_REGFILE(IURDATA)'     ;shall be placed in the upper rfile
.lnkcmd '-S ROM(CODE_??CLIB)'           ;shall be placed in ROM
.lnkcmd '-S ROM(CONST_??CLIB)'          ;shall be placed in ROM
.lnkcmd '-S ROM(??INIT_FUNC_PTR_TABLE)' ;shall be placed in ROM
.lnkcmd '-S ROM(??PTSVECTORS)'          ;shall be placed in ROM
.lnkcmd '-S ROM(IRDATA_INI)'            ;shall be placed in ROM
.lnkcmd '-S ROM(IURDATA_INI)'           ;shall be placed in ROM
.lnkcmd '-S ROM(??INIT_DATA_SEGMENT_ADDRESSES)' ;shall be placed in ROM
.lnkcmd '-S ROM(??DATA_SEGMENT_ADDRESSES)' ;shall be placed in ROM
;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;
; Startup code follows...
;
.EXTRNF main
.rseg STACK,mem
.ds 40H                               ;default stack size
.rseg CODE_??CLIB
.FUNC ?START
.PUBLIC ?START
        ld     AX,#.sfe STACK
        add    AX,#2
        and    AX,#0FFFEh
        ld     SP,AX
.rseg CONST_??CLIB
??STACK_BOTTOM_ADDRESS .dcw (.sfb STACK)
??STACK_TOP_ADDRESS   .dcw (.sfe STACK)
.public ??STACK_BOTTOM_ADDRESS,??STACK_TOP_ADDRESS
.rseg CODE_??CLIB
.IF .MODEL == 2
        ecall  main
.ELSE
        lcall  main
.ENDIF
        sjmp  $
.ENDF
.end

```

Important note: This version of C startup initializes the stack only and then jumps directly to the `main()`. This does not guarantee a valid behavior of the MCC96 library, because the proper initialization of static resources and calls to initialization routines is NOT performed. Therefore, the usage of 'truncated' startup is NOT RECOMMENDED STRONGLY. Also note that you may completely omit ALL THE STARTUP CODE, but the resulting program will not compile (i.e. link) unless you write your own C startup code in accordance to rules described in our documents.

3. General Application Tips

3.1. Starting Address

You do not have to specially setup your code to start at 2080H. If linking options are specified correctly, this will be done automatically.

3.2. Stack size

Some of the frequently asked questions are:

- > *Do your tools calculate the stack requirements for an entire linked*
- > *application based upon the worst case function call tree(s)? Do the*
- > *tools calculate the stack required for each individual function?*

The answer for both is: "No". We are not using compiled stack and call tree(s). The compiler uses traditional stack approach. The merit concerns our compiler that fully supports function reentrancy and recursion, which are **really** required for compliance to the C Standard and are **very** important for interrupt-intensive applications. Note that the concept "*worst case function call tree*" is not applicable for programs that use recursion and interrupts. Also note that the "*compiled stack*" approach is only effective when the size of stack is smaller than the size of on-chip RAM (200-1500 bytes). Traditional stack architecture is more effective when the stack is located in the external memory, because traditional stack uses short-indexed addressing whereas compiled stack uses long-indexed addressing.

Required stack depth for applications that use recursion can not be calculated statically (during compilation) in any way. If you are not sure about stack depth, you can use the `#pragma prolog` directive to perform runtime stack overflow check.

3.3. Flash Programming Application Note

Flash programming can be done using assembly language. Here are some guidelines followed by the source code example.

1. Write low-level programming routines in assembly language. These routines must be "relocatable". Thus, DO NOT use `BR` instructions in assembly routines (this is quite easy to do). Place programming routines in the separate segment (i.e. `FL_CODE`). This segment will be located in the FLASH memory (ROM address area). Remember that FLASH memory is not functional during programming, therefore interrupts can not be used in low-level flash programming routines.
2. Write some assembly code that will copy `FL_CODE` to the RAM at the program startup phase.
3. Reserve space for `FL_CODE` segment in the RAM address area. Reserve certain quantity, for example, 256 bytes (that was quite sufficient for the example below). Use an absolute segment for this purpose - you have to know absolute addresses to call "relocated" low-level assembly functions.
4. Write C header with interface to the assembly routines. Now you can use routines in any C program.

Follow the sample code for our MCC-96/MCA-96 toolset.

===FLPRG.MCA===

```

; Low-level flash programming routines
;
; Note 1. This code is intended for word-organized flash memory.
; Initially, it was developed for Intel 28F400BV-T. Code for
; byte-organized flash will be simpler.
;
; Note 2. This code is suitable for all MCC-96 memory models)
;
; (c)1998-1999 PHYTON
;
; Declarations of standard temp regs
;
.include 'tmpreg.inc'

; Error codes
;
; routines can return the following error codes
;(0 - no error)
FL_ERR_ERASE_vpp_range .equ 2 ;erase: vpp range error
FL_ERR_ERASE_cmd_sequence .equ 3 ;erase: cmd sequence error
FL_ERR_ERASE_block_erase .equ 4 ;erase: block erase error
FL_ERR_PROGR_vpp_range .equ 6 ;program: vpp range error
FL_ERR_PROGR_word_program .equ 7 ;program: word program error

; Flash memory commands
;
FL_ERASE_CMD .equ 0ff20h ;erase setup cmd
FL_ERCNF_CMD .equ 0ffd0h ;erase confirm cmd
FL_RDSCR_CMD .equ 0ff70h ;read status register cmd
FL_RDARR_CMD .equ 0ffffh ;read array cmd
FL_PROG_CMD .equ 0ff40h ;program byte/word cmd
FL_CLRSR_CMD .equ 0ff50h ;clear status register cmd

; Definitions of global variables
; (used for passing parameters to assembly routines)
;
.rseg RDATA,reg
.if .model .eq 0
.align 1
FLaddr .ds 2 ;address of location to be programmed
;or address of memory block to be cleared
.type FLaddr(.uchar_ptr)
.word FLaddr
.else
.align 2
FLaddr .ds 4 ;address of location to be programmed
;or address of memory block to be cleared
.type FLaddr(.uchar_ptr)
.dword FLaddr
.endif

FLdata .dsw ;data to be programmed
FLerrno .dsw ;holds error code (0 - no error)
.public FLaddr
.public FLdata
.public FLerrno

; Reserve space in RAM for low-level assembly routines
;
.aseg FL_CODE_RAM,mem
FL_CODE_RAM_LABEL .ds 256 ;reserve space for copy of code
.lnkcmd '-N RAM(08000H-080FFH)' ;to prevent overlapping of this segment
;by relocatable segments
.lnkcmd '-S RAM(FL_CODE_RAM)' ;places the segment in the RAM address area

```

```

////////////////////////////////////
; Low-level assembly routines follows...
;
.rseg FL_CODE,mem
.lnkcmd '-S ROM(FL_CODE)'      ;places the segment in the ROM address area
;jump table at start of segment
FUNCTION_ENTRY .macro fname
.align 2      ;align at word boundary
.func fname .void (.void)
SJMP __ & fname ;goto actual function
.endf
.endmac

FUNCTION_ENTRY FL_EraseBlock      ;offset 0 in the FL_CODE segment
FUNCTION_ENTRY FL_WriteWord      ;offset 4 in the FL_CODE segment
FUNCTION_ENTRY FL_WriteByte      ;offset 8 in the FL_CODE segment
FUNCTION_ENTRY FL_Reset          ;offset 12 in the FL_CODE segment

;these macros provide usability of code in all memory models
_XOP .macro op,a,b
.if .model .eq 0
op a,b
.else
E & op a,b
.endif
.endmac

XLD .macro a,b
_XOP LD,a,b
.endmac
XLDB .macro a,b
_XOP LDB,a,b
.endmac
XST .macro a,b
_XOP ST,a,b
.endmac
XSTB .macro a,b
_XOP STB,a,b
.endmac

;erase flash block at FLaddr
.func __FL_EraseBlock
    ld    ax,#FL_ERASE_CMD
    XST   ax,[FLaddr]
    ld    ax,#FL_ERCNF_CMD
    XST   ax,[FLaddr]
_rdsr:
    XLDB  bl,[FLaddr]
    jbs   bl,7,_ready
_ready:
    jbc   bl,3,_vpp_ok
    ld    FLerrno,#FL_ERR_ERASE_vpp_range
    ret
_vpp_ok:
    andb  al,bl,#00110000b
    cmpb  al,#00110000b
    jne   _seq_ok
    ld    FLerrno,#FL_ERR_ERASE_cmd_sequence
    ret
_seq_ok:
    jbc   bl,5,_erase_ok
    ld    FLerrno,#FL_ERR_ERASE_block_erase
    ret
_erase_ok:
    clr   FLerrno
    ld    ax,#FL_RDARR_CMD
    XST   ax,[FLaddr]
    ret
.endf

```

```

;write word FLdata to address FLaddr
.func __FL_WriteWord
    ld    ax,#FL_PROG_CMD
    XST  ax,[FLaddr]
    XST  FLdata,[FLaddr]
_rdsr:
    XLDB bl,[FLaddr]
    jbs  bl,7,_ready
_ready:
    jbc  bl,3,_vpp_ok
    ld   FLerrno,#FL_ERR_PROGR_vpp_range
    ret
_vpp_ok:
    jbc  bl,4,_progr_ok
    ld   FLerrno,#FL_ERR_PROGR_word_program
    ret
_progr_ok:
    clr  FLerrno
    ld   ax,#FL_RDARR_CMD
    XST  ax,[FLaddr]
    ret
.endf

.func __FL_Reset
    ld   ax,#FL_RDARR_CMD
    XST  ax,[FLaddr]
    XST  ax,[FLaddr]
    ld   ax,#FL_CLRSR_CMD
    XST  ax,[FLaddr]
    ld   ax,#FL_RDARR_CMD
    XST  ax,[FLaddr]
    ret
.endf

;write low byte of FLdata to address FLaddr
.func __FL_WriteByte
    push FLdata
    push .word FLaddr
    or   FLdata,#0ff00h ;mask unneeded byte
;note:
; flash memory has word organization.
; low bytes located at even addresses.
; high bytes located at odd addresses.
; LSB of address has to be cleared.
    jbc  .byte FLaddr,0,_0
    andb .byte FLaddr,#0feh
    xchb .byte FLdata,.byte FLdata+1
_0: scall __FL_WriteWord
    pop  .word FLaddr
    pop  .word FLdata
    ret
.endf

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; To copy routines from flash to RAM
; we are using regular C startup methods
;
.rseg ??INIT_DATA_SEGMENT_ADDRESSES,mem

_DCX .macro param
.if .model .eq 0
.dcw param
.else
.dcd param
.endif
.endmac

_DCX <.SFB FL_CODE_RAM > ;copy to: starting address
_DCX <.SFE FL_CODE_RAM > ;copy to: ending address
_DCX <.SFB FL_CODE > ;copy from: starting address

.END

```

===END OF FLPRG.MCA===

===MAIN.C===

```

/*
*****
* FLASH programming sample code
*
* (c)1998 PHYTON
*****
*/

#include <other.h> /* prototypes for _ei_() and _di_() */

/*
*****
* /START/ INTERFACE TO THE LOW_LEVEL ASM-ROUTINES /START/
*
*****
* NOTE: RAM for flash programming code MUST BE LOCATED AT 0x8000
* NOTE: RAM for flash programming code MUST HAVE SIZE AT LEAST 256 BYTES
*
*/

// global variables used to pass parameters to
// low-level asm-routines
#pragma datloc reg
extern char * FLaddr; /* flash memory address for erase/write operations */
extern unsigned FLdata; /* data to be write to flash */
extern unsigned FLerrno; /* error code; 0 - no error */
#pragma datloc .

//macros that call corresponding low-level asm-routines
typedef void (*FL_void_fp)(void);
#define FL_FUNC_BASEADDR 0x8000
#define FL_EraseBlock (*(FL_void_fp)((FL_FUNC_BASEADDR) + 0 ))
#define FL_WriteWord (*(FL_void_fp)((FL_FUNC_BASEADDR) + 4 ))
#define FL_WriteByte (*(FL_void_fp)((FL_FUNC_BASEADDR) + 8 ))
#define FL_Reset (*(FL_void_fp)((FL_FUNC_BASEADDR) + 12))
/*
*****
* /END/ INTERFACE TO THE LOW_LEVEL ASM-ROUTINES /END/
*
*****
*/

//declarations of "wrappers" (local static functions)
static int EraseFlashBlock(void * addr);
static int WriteFlashByte(void * addr, char data);

void main(void)
{
    if (!EraseFlashBlock(0)) /* ERASE PAGE 0 OF FLASH ROM */
        goto flash_error;
    /* PROGRAM BYTE 0x55 TO */
    /* FLASH LOCATION AT 0x100 */
    if (!WriteFlashByte((void*)0x100,0x55))
        goto flash_error;

flash_error:
    /* handle error here... */
    while(1); /* lock up if error 8-) */
}

static int EraseFlashBlock(void * addr)
{
    FLaddr = addr;
    _di_();
    FL_EraseBlock();
    _ei_();
    return !FLerrno;
}

static int WriteFlashByte(void * addr, char data)
{
    FLaddr = addr;
    FLdata = data;
    _di_();
    FL_WriteByte();
    _ei_();
    return !FLerrno;
}

```

===END OF MAIN.C===

3.4. Understanding Memory Map

Question:

>I have looked at the MAP file and it doesn't appear to have
>organized things in the proper place (or maybe I'm not understanding how to
>read it). ?

The best way to explore memory layout is to read the

```
* * * ADDRESS AREAS MAP * * *
```

section of the mapfile.

3.5. Intel RISM and Phytion Tools

Question:

>I am using an Intel Evaluation board (EV80C196KD) for my development. This board
>comes with a "RISM" monitor/debug program that requires certain areas to be
>left for its use only. The Phytion startup code is placing information into
>areas of the 80C196 address space that cause the debugger problems
>(specifically, the write to the NMI interrupt vector - although there are
>other areas that I haven't been able to nail down yet).

To use RISM, you have to include customized assembly startup in your program instead of standard one.

First, include (assemble and link) the RSV_RISM.MCA file to your project. It redefines several MCC-96 library modules in such a way that the resulting program does not initialize CCBs, reserved memory locations by OFFH's, and does not set up the NMI, TRAP and UNIMPLEMENTED OP CODE interrupt vectors. In addition, it reserves locations 030H-04EH in the lower register file for RISM (required; see RISM manual).

Second, link C96 library with the -o prefix. Fragment of sample of linker command file:

```
...  
mymodule1  
mymodule2  
mymodule3  
RSV_RISM.MCO #RISM support module.  
-o C96S #MCC96 library: set priorities of all modules to "library";  
 #RISM support module will override standard assembly startup.  
...
```

>One last question. How do I setup the project
>window so that it contains the '-o C96S' option?

PDS-96: Click the "Project" item in the main menu. Select "Tools setup". Select "Linker" in the "Individual tools setup" list. Press "Edit" button. CLEAR the "Use standard library" checkbox. In the "Other options" text box, type "-o C96S.MCL". Press "OK". Press "Done".

Customized assembly startup (RSV_RISM.MCA) follows here (changes have gray background).


```
.lmodule2 ??DEFAULT_INT02MODULE
.rseg CSEG_??ISR,mem
?INTERRUPT02: di
    sjmp $
.public ?INTERRUPT02
.endmod
```

```
.lmodule2 ??DEFAULT_INT03MODULE
.rseg CSEG_??ISR,mem
?INTERRUPT03: di
    sjmp $
.public ?INTERRUPT03
.endmod
```

```
.lmodule2 ??DEFAULT_INT04MODULE
.rseg CSEG_??ISR,mem
?INTERRUPT04: di
    sjmp $
.public ?INTERRUPT04
.endmod
```

```
.lmodule2 ??DEFAULT_INT05MODULE
.rseg CSEG_??ISR,mem
?INTERRUPT05: di
    sjmp $
.public ?INTERRUPT05
.endmod
```

```
.lmodule2 ??DEFAULT_INT06MODULE
.rseg CSEG_??ISR,mem
?INTERRUPT06: di
    sjmp $
.public ?INTERRUPT06
.endmod
```

```
.lmodule2 ??DEFAULT_INT07MODULE
.rseg CSEG_??ISR,mem
?INTERRUPT07: di
    sjmp $
.public ?INTERRUPT07
.endmod
```

```
;RISM .lmodule2 ??DEFAULT_INT16MODULE
;RISM .rseg CSEG_??ISR,mem
;RISM ?INTERRUPT16: di
;RISM     sjmp $
;RISM .public ?INTERRUPT16
;RISM .endmod
```

```
;RISM .lmodule2 ??DEFAULT_INT17MODULE
;RISM .rseg CSEG_??ISR,mem
;RISM ?INTERRUPT17: di
;RISM     sjmp $
;RISM .public ?INTERRUPT17
;RISM .endmod
```

```
.lmodule2 ??DEFAULT_INT08MODULE
.rseg CSEG_??ISR,mem
?INTERRUPT08: di
    sjmp $
.public ?INTERRUPT08
.endmod
```

```
.lmodule2 ??DEFAULT_INT09MODULE
.rseg CSEG_??ISR,mem
?INTERRUPT09: di
    sjmp $
.public ?INTERRUPT09
.endmod
```

```
.lmodule2 ??DEFAULT_INT10MODULE
.rseg CSEG_??ISR,mem
?INTERRUPT10: di
    sjmp $
.public ?INTERRUPT10
.endmod
```

```

.module2 ??DEFAULT_INT1MODULE
.rseg CSEG_??ISR,mem
?INTERRUPT11: di
    sjmp $
.public ?INTERRUPT11
.endmod

.module2 ??DEFAULT_INT12MODULE
.rseg CSEG_??ISR,mem
?INTERRUPT12: di
    sjmp $
.public ?INTERRUPT12
.endmod

.module2 ??DEFAULT_INT13MODULE
.rseg CSEG_??ISR,mem
?INTERRUPT13: di
    sjmp $
.public ?INTERRUPT13
.endmod

.module2 ??DEFAULT_INT14MODULE
.rseg CSEG_??ISR,mem
?INTERRUPT14: di
    sjmp $
.public ?INTERRUPT14
.endmod

;RISM HERE .module2 ??DEFAULT_INT15MODULE
;RISM HERE .rseg CSEG_??ISR,mem
;RISM HERE ?INTERRUPT15: di
;RISM HERE     sjmp $
;RISM HERE .public ?INTERRUPT15
;RISM HERE .endmod

.pmodule ??ISR_TRANSIT_MODULE
;
;FIX to enable ISRs to be allocated at any page for LARGE memory model
;
ISR_JUMP_TO_ANY_PAGE .MACRO NN
.EXTRN (mem) ?INTERRUPT&NN
??_TRANS_?INTERRUPT&NN&: EJMP ?INTERRUPT&NN
.PUBLIC ??_TRANS_?INTERRUPT&NN
.ENDMAC
;
;.EXTRN (mem) ?INTERRUPT00
?INTERRUPT00__: ejmp ?INTERRUPT00
;.PUBLIC ?INTERRUPT00__
;
; and so on...
;
;                                     ;(18 * 4 = 72 bytes)
;.IF .MODEL == 2                       ;FIX for problem w/ISRs for LARGE model
.lnkcmd '-N ROM(0FF8004h-0FF804Bh)' ;reserve space in ROM for ??ISR_TRANSIT_SEG
.lnkcmd '-S ROM(??ISR_TRANSIT_SEG)' ;to be placed in the ROM address area
.aseg ??ISR_TRANSIT_SEG,mem
.org 0FF8004h
ISR_JUMP_TO_ANY_PAGE 00
ISR_JUMP_TO_ANY_PAGE 01
ISR_JUMP_TO_ANY_PAGE 02
ISR_JUMP_TO_ANY_PAGE 03
ISR_JUMP_TO_ANY_PAGE 04
ISR_JUMP_TO_ANY_PAGE 05
ISR_JUMP_TO_ANY_PAGE 06
ISR_JUMP_TO_ANY_PAGE 07
ISR_JUMP_TO_ANY_PAGE 08
ISR_JUMP_TO_ANY_PAGE 09
ISR_JUMP_TO_ANY_PAGE 10
ISR_JUMP_TO_ANY_PAGE 11
ISR_JUMP_TO_ANY_PAGE 12
ISR_JUMP_TO_ANY_PAGE 13
ISR_JUMP_TO_ANY_PAGE 14
;RISM HERE ISR_JUMP_TO_ANY_PAGE 15
;RISM HERE ISR_JUMP_TO_ANY_PAGE 16
;RISM HERE ISR_JUMP_TO_ANY_PAGE 17
.ENDIF                                ;FIX for problem w/ISRs for LARGE model
.endmod

.pmodule ??RESERVED_FAKE_END_MODULE
.end

```