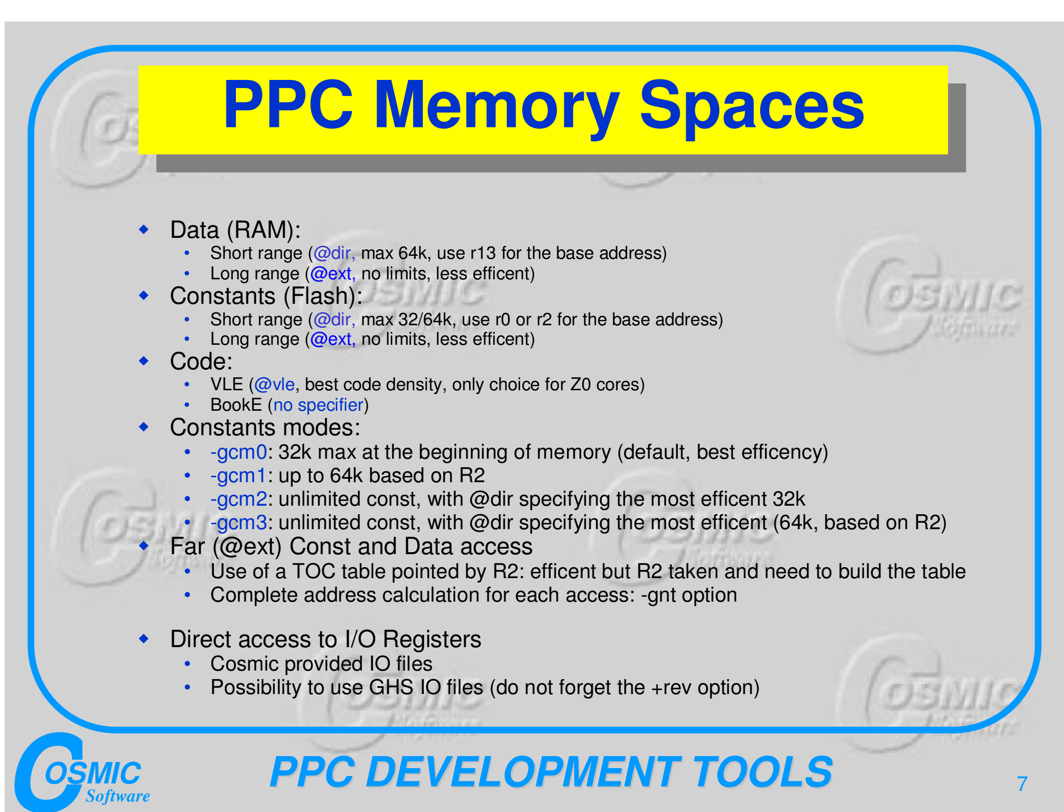
Getting started with the PPC tools:

compiler options and linker file

This Application Note describes how to best use the Cosmic Toolchain for PPC depending on the derivative used (how much RAM and Flash are available) and the application layout (how much code / constants / ram are required). The main focus is on choosing the right compiler options and writing (customizing) the linker and startup files accordingly.

Important note: some of the features described in this application note only apply to compiler version 4.2.8 (November 2010) and higher.

Let us first review briefly the "Memory Spaces" of the compiler, that is, the different ways that the compiler can use to access code or data:



The executable code can be VLE or BookE: this is often defined once for all for a given chip (although some bigger chips can mix the two modes by configuring properly the Memory Management Unit) and we will not analyse it any further in this document.

More interesting for this Application Note, are the different "ways" to access data (including constants): it basically comes down to build the full 32-bits address of every variable or use an index register. The first method is more general (no limitations on the size, number, or placement of the data), while the second is more efficent, as you can see in the following example:

```
@ext int extvar;        // will be accessed building its full (32 bits) address (8 bytes, 3 cycles)
@dir int dirvar;        // will be accessed with an index register (6 bytes, 2 cycles)
void prova(void)
{
        extvar = 3;
        dirvar = 5;
}
```

```
3556              ; 33        extvar = 3;
3558  0000 50800000   lwz    r4,L6           // load the variable address
3559  0004 4833       li     r3,3
3560  0006 d034       stw    r3,0(r4)        // write the variable
3561              ; 34        dirvar = 5;
3563  0008 4853       li     r3,5
3564  000a 546d0000   stw    r3,_dirvar(r13) // the variable address is an offset + base register
```

Selecting the memory model

Once this is clear, we can proceed to define a Memory Model: *a Memory Model is a compiler option that tells the compiler how to make some default choices, like what code to produce (VLE or BookE), where to place variables (@dir or @ext) and constants (@dir/@ext, indexed by r0 or r2) and how to manage floating point (using hardware or simulating with libraries)*

The table below shows the memory models available (note that some of them only exist since v4.2.8, relased in November 2010)

Model	Instr.	Data	Constants	Floating Point
+modv	VLE	64K (r13)	32K (r0)	Libraries
+modvc	VLE	64K (r13)	64K (r2)	Libraries
+modvf	VLE	64K (r13)	32K (r0)	Hardware
+modvfc	VLE	64K (r13)	64K (r2)	Hardware
+modvl	VLE	Extended	Extended	Libraries
+mods	Book E	64K (r13)	32K (r0)	Libraries
+modsc	Book E	64K (r13)	64K (r2)	Libraries
+modsf	Book E	64K (r13)	32K (r0)	Hardware
+modsfc	Book E	64K (r13)	64K (r2)	Hardware
+modl	BookE	Extended	Extended	Libraries

A few notes about memory models:

- the memory models only cover the more common situations: as you can see, there is no memory model (for example) that put variables @ext AND produce VLE code by default: if this is needed, experienced users can create and use their own memory models

- the memory model assigns a space only to those variables that do not have it specified in the source. For example:

```
@ext int a;     // will go to .bss (==be accessed using the 32bit address) regardless of the memory model
int b;          // will go to .bss with +modl, to .sbss (fast access) with any other model
```

At this point it should be quite easy to select the best memory model for a given application: *the smallest memory model that fits the application will be the best* (because the code produced with this memory model will be smaller and faster).

Examples:

1) for a “small” application (<64kb data, <32kb of constants, no code size limit), you should select the +modv (VLE) or +mods (BookE) memory model.
2) For a “medium size” application (<64kb data, <64kb of constants, no code size limit), you should select the +modvc (VLE) or +modsc (BookE) memory model. If the application use floating point, and if hardware floating point is supported by the chip, select +modvfc (VLE) or +modsfc (BookE) for a better efficency of floating point operations.
3) For a “big” application (>64k data and/or constants), you should use the +modl memory model, so that variables will be @ext by default. Note however, that, in this way, ALL the variables will be @ext and therefore will be accessed in non-optimal

way. In order to improve performance, you can split your variables in such a way that some of them (max 64k, possibly the most used ones) are @dir, while the rest (no size limitation, possibly the rarely-used ones) are @ext. In order to achieve this, there is some manual modification needed, and you can do it in two ways: either you start with a +modv (or other model where the variables are @dir by default) and declare some variables @ext by hand (you need to do this as long as the remaining @dir variables fit into 64k) or you start with no model (or another model where the variables are @ext by default) and you "force" some variables @dir by hand (no more than 64k, but the more you are close to this limit, the more optimized your application will be).

Note that the smallest chips in the PPC family are VLE-only and do not have more than 64k RAM or hadware floating point support, therefore the choice of memory models is reduced to +modv or +modvc.

Memory models and Startup Files

The "startup" is a bit of code that is executed just after reset and before the main program. Its purpose is to setup some "resources" (initialize the Hardware/RAM, the stack pointer and so on) that will be used later on (see the end of this document for a detailed description of a typical startup file).

In most compilers, at least two different startup files exist: one for the case where initialized variables are used, and one, simpler and faster, for the case where variables are not initialized. In addition to this, in the case of the PowerPC compiler, two more conditions must be taken into account:

- r2 needs to be initialized to point at constants or not
- the startup code must be VLE or BookE

According to the conditions above, you must use the correct startup file for your application by choosing it in the table shown in the next page.

Note that all these pre-compiled versions of the startup file actually come from the same source (commented later in this document) compiled with different #define: if you need a special startup file just copy and modify the one provided with the compiler, taking care of compiling it with the proper defines.

Models	Without Data Initialization	With Data Initialization
+modv, +modvf	crtsv.ppc	crtsiv.ppc
+modvc, +modvfc	crtsvc.ppc	crtsivc.ppc
+mods, +modsf	crts.ppc	crtsi.ppc
+modsc, +modsfc	crtsc.ppc	crtsic.ppc

Writing the linker file

The "linker file" is an input file to the linker that tells it where and how to place all the different parts of your application (code, data, constants) and what kind of checks to do on them.

The linker file must be written/customized to include information about:

- the derivative you want to run your application on (FLASH and RAM sizes..)
- the compiler options used (need or not to initialize some registers, like R2 for accessing big constants), use of the +split options in order to discard unused code..
- some "features" of the application (use or not of initialized variables, use of user-defined sections to be placed at special addresses...)

The linker file is based on *segments* (a segment is an unbreakable piece of information of the same kind), therefore, before looking at a linker file, we must know what kind of information the compiler will put in predefined segments (actually the Compiler + Assembler generate *sections* and the linker combines several sections of the same kind into a segment, see the User Manual for more details):

.text	executable code (BookE mode)
.vtext	executable code (VLE mode)
.const	text string and constants (32 bits access)
.sconst	@dir constants (optimized access based on a register)
.data	initialized variables in RAM (@ext, 32 bits access)
.bss	uninitialized variables in RAM (@ext, 32 bits access)
.sdata	initialized variables in short range (@dir, optimized access based on R13)
.sbss	non-initialized variables in short range (@dir, optimized access based on R13)

These are the standard segments that a typical linker file will have to consider, but note that the list might be shorter or longer:

- some of the standard segments are sometimes not generated (for example .data and .bss are not generated in the +modv memory model if you don't declare any @ext variable by hand)
- some user segments may be added: even if you don't add any of your own you are very likely to see the .rchw and .vector segments that are used in all Cosmic examples to place the Boot Assist Module and the Interrupt Vectors at suitable addresses in memory

Note also that some of the segments have a "default placement": that means that, even if you forget to specify such a segment in your linker file, the linker will put it in a standard place (typically attached to another segment) and you will get no linker error: it is not suggested to rely on this feature; it is better to specify explicitly all the segments in the linker file.

Let's now see an example of a real linker file, written for a Pictus device with 512k Flash and 40k RAM, and configured for the use of >32kb constants (based or r2) and dead code removal (+split compiler options).
The file is as follows:

```
# Link command file for Pictus 512k Flash and 40k RAM
# 4k BAM+vectors, 444k code, 64k consts
# Cosmic Software

+seg .rchw -b 0 -m 0x4000 -n rchw -k                        # bam start address
+seg .vector -a rchw -n vect -r 11 -k                       # vectors
+seg .vtext -b 0x4000 -m 0x6c000 -n vtext -it -r 2          # code
+seg .sconst -b 0x70000 -o 0x8000 -m 0x10000 -n sconst -r 2 # constants
+seg .sdata -b 0x40000000 -o0x8000 -n sdata -r 2 -id        # data
+seg .sbss -a sdata -n sbss                                 # uninitialized data
+def __sbss=pstart(sbss)                                    # start address of bss

crtsivc.ppc
bam.o
vec560p.o
<< list of the application objet files >>
"C:\Program Files\COSMIC\CXPPC\Lib\libiv.ppc"               # libraries
"C:\Program Files\COSMIC\CXPPC\Lib\libmv.ppc"               # libraries
```

```
+def __sram=pstart(sdata)            # start address of bss
+def __memory=pend(sbss)             # symbol used by library
+def __stack=0x4000A000              # stack pointer initial value
+def __sdata=0x40008000              # data pointer value
+def __eram=__stack                  # ram end address
+def __sconst=0x78000                # constants start address
```

As you can see, most of the lines define segments (+seg) or symbols (that will be used in the startup file, explained in the next chapter), or object files (only 3 often used object files are specified here, but the list can be much longer, typically as long as the number of source files in the application).

Let's comment the lines one by one in order to see what they do and what happens if we change some of the parameters they contain (that is, some typical errors to avoid):

```
+seg .rchw -b 0 -m 0x4000 -n rchw -k          # bam start address
```

defines a segment named "rchw" (without the dot, it's the parameter of the –n option) that will contain section(s) of the type .rchw (in the Cosmic examples, this section is used to produce a single costant, the BAM value). This segment will be located at address 0 (as specified by the –b parameter and as required by the hardware) and its maximum size (including the size of all segments attached to it, see the next line) will be of 0x4000 bytes (that is, the link will give an error of type "segment overflow" only when this limit is reached). The –k parameter tells the linker that this segment is a "root" in the call tree of the application: when the dead code removal is enabled with the +split compiler option, segments with the –k option are kept (not discarded) even if they are not called from anywhere else in the application.

```
+seg .vector -a rchw -n vect -r 11 -k         # vectors
```

defines a segment named "vect" (note that the name can be omitted if it is not used anywhere else in the linker file) that will contain section(s) of the type .vector (in the Cosmic examples, this section is used to produce a table of constants, the Interrupt Vectors). This segment will not be located at an absolute address (no –b option), but rather be attached (-a option) to the segment named "rchw". This means that the starting address of this segment depends on the size of the previous one. Typically the vectors should therefore start at address 0x4, but the –r11 parameter make sure that this address is aligned on at least 11 bits (as required by the

hardware), so the vectors are actually located at address 0x800, as you can easily verify in the map file:

```
start 00000800 end 00001000 length  2048 segment vect
```

Note that this segment is marked with the –k option as well: if you forget this parameter, the interrupt routines will be removed by the linker, because they are not "called" from anywhere in the main flow.

The rule to follow for the dead code removal is quite simple in the end: use the +split option on the compiler and put –k on the BAM and Vectors segments.

```
+seg .vtext -b 0x4000 -m 0x6c000 -n vtext -it -r 2                    # code
```

defines a segment named "vtext" that will contain sections of the type .vtext . This segment will contain the executable code, or, to be precise, the VLE part of the executable code. Code will start at address 0x4000 (an arbitrary value chosen to leave some space after the vectors "just in case" that can be still addressed by r0) and its maximun size will be 0x6c000 (444kb). This value depends both on the derivative chosen (how much FLASH is available) and the memory layout (how much place do you leave for constants and where you place them). If you calculate it wrongly, several kind of errors can happen:

- if you put a number smaller than the real, the linker will complain that there is no more memory ("segment overflow" message) even when this is not true
- if you put a number bigger than real, in this specific case you will have a linker message saying that segment "text" and "sconst" overlap, but in another typical case, where constants are just appended after the code instead of being located at an absolute address, you can have far more strange behaviours, as the linker produces a code bigger than the available memory without any error or warning

Note also the –it flag that tells the linker that the initialization table (a table used by the startup file to copy the initial values of initialized variables from ROM to RAM) should be attached to this segment.

The –r2 parameter tells the linker that this segment must be aligned on a least 2 bits: actually one bit (-r1) would be enough for VLE code, but since VLE and standard code can be mixed on some derivatives, it is better to leave –r2 on all executable segments.

```
+seg .sconst -b 0x70000 -o 0x8000 -m 0x10000 -n sconst -r 2          # constants
```

defines a segment named "sconst" that will contain sections of the type .sconst . This segment will contain the constants that are declared @dir, either implicitly (all memory models) or explicitly. The constants in this segment will be located at address 0x70000, that is, in the last (highest) 64k of memory for the device we are using. The –m parameter tells that the max size of this segment is 64k: if we do not specify this, strange (and difficult to debug) things can happen: the linker will not complain for @dir constants >64k, but since the addressing is only on a 16 bits offset, you will end up with some constants overlapping each other. The –o parameter sets the logical start address: for the PPC this value is always 0x8000 because the offsets used in the load and store instructions are signed.

```
+seg .sdata -b 0x40000000 -o0x8000 –m0xA000 -n sdata -r 2 -id                    # data
```

defines a segment that will contain the initialized data (coming from C declarations like int myvar = 10;) declared @dir, either implicitly (+modv or similar memory models) or explicitly. The segment will start at 0x40000000 (this is forced by the hardware and could vary between one derivative and another), and have a maximum size including whatever is attached to it (in our case the .sbss -> all variables, initialized + non initialized), of 40k. This –m0xA000 parameters is worth some considerations:

- we have set this value to the maximum size of RAM available: this means that the linker will not complain until we reach 40k, but the application will stop working before that, because the stack, which is placed at the end of the memory and grows backwards, will overlap with the variables (which leads to seemly random, difficult to debug application crashes)
- Another choice could have been to set the limit lower, so that we are sure to leave at least some place for the stack: this is possibly a good idea, but since the stack size is not known at compile time anyway (although it can be estimated under some restrictions) it's difficult to set a meaningful value.
- This value should never be set bigger than 0xFFFF (even if some micros has a RAM bigger than that), since this segment is address with a 16 bits offset; see the explanation for segment .sconst

The –id parameter tells the linker that this segment contains initialized data: if you forget to specify it, your data will not be initialized at startup and will contain random values (at best all zeros) at the beginning of the application.

```
+seg .sbss -a sdata -n sbss                              # uninitialized data
```

defines a segment that will contain the non initialized data (coming from C declarations like int myvar;) declared @dir, either implicitly (+modv or similar memory models) or explicitly. This segment is appended to the previous one, so we don't know its start address before linking (this information can be found in the map file). Note that all features and checks of this segment (alignment, maximum size..) are inherited from the segment it is appended to.

```
+def __sbss=pstart(sbss)                                 # start address of bss
```

defines a symbol named _sbss that will be used somewhere else (typically in the startup file, see next chapter). In this case the symbol is set to the value of the start of the segment named "sbss" and is used to force to zero all the non-initialized variables.

```
crtsivc.o
bam.o
vec560p.o
<< list of the application objet files >>
```

This is the list of the object files that must be linked. The first 3 are pretty standard but can vary (especially the startup file: several versions exist even in you stay within the Cosmic-provided libraries), while the rest is completely application dependent. The order is not important. Note that the list of files can be redirected to an external text file for easying automatic builds of similar configurations.

```
"C:\Program Files\COSMIC\CXPPC\Lib\libiv.ppc"            # libraries
"C:\Program Files\COSMIC\CXPPC\Lib\libmv.ppc"            # libraries
```

Specify the libraries to be linked. A few notes:

- libraries must be linked after the object files
- the order of libraries is important
- the number of libraries depends on the complexity of the application
    - libm : always needed
    - libi : almost always needed (when doing complex operations on integers)
    - libf: (not linked in this example) needed only when using floating point
- different memory models require different libraries: see the UM for details

```
+def __sram=pstart(sdata)          # start address of bss
+def __memory=pend(sbss)           # symbol used by library
+def __stack=0x4000A000            # stack pointer initial value
+def __sdata=0x40008000            # data pointer value
+def __eram=__stack                # ram end address
+def __sconst=0x78000              # constants start address
```

define some symbols that will be used in the startup file, see the next chapter.

Customizing the startup file

The startup file contains code that is executed immediatly after the reset and before the application starts. A typical startup code, as provided by Cosmic both in source and object formats, will do the following

- Initialize some registers (mainly the stack pointer)
- Initialize variables if required (by copying their initial value from ROM to RAM)
- Clear (set to zero) the non-initialized variables (and/or all the rest of the memory)
- Jump to main

The startup file must be modified/customized in the following cases:

- you are using a special configuration (set of compiler options and/or linker file) for which Cosmic does not provide a pre-compiled startup
- you write your own startup because you want/need special features and/or you have strict timing requirements (example: no time to zero all the memory before starting the application)

If you need to modify the startup file, it is important to understand how it works and how it is related to the linker file: the linker file defines some values that are used in the startup, as can be seen in the example below, showing the startup file that goes with the linker file explained in the previous chapter:

Note: this startup file is not one of those provided by Cosmic in the compiler package (although it is very similar to them): if you want to write your own startup file please copy and then modify the startup that is in the compiler package.

```
;	Sample C Startup code with data initialization and R2 initialization to address 64k constants
;	Copyright (c) 2010 by COSMIC Software
;
	xdef	_exit, __stext
	xref	_main, __memory, __sdata, __sbss, __stack, __sconst, __idesc__, __eram
;
ifdef VLE
	vle	on
.vtext:		section	.text
endif
```

```
;
atab:
      dc.l    __sdata            ; init value of data pointer
      dc.l    __sram-4           ; start of ram to clear
      dc.l    __eram-4           ; end of ram to clear
      dc.l    __stack            ; init value of stack pointer
      dc.l    __idesc__          ; descriptor start address
      dc.l    __sconst
__stext:
      lis     r3,.atab           ; get table
      addi    r3,.atab           ; address
      lwz     r4,4(r3)           ; get start of ram
      lwz     r5,8(r3)           ; get end of ram
      sub.    r5,r4              ; byte size
      beq     init               ; empty, skip
      srwi    r5,2               ; word size
      li      r0,0               ; to clear the bss
      mtctr   r5                 ; set counter
zbcl:
      stwu    r0,4(r4)           ; clear memory
      bdnz    zbcl               ; count down and loop back
init:
      lwz     r1,12(r3)          ; initialize SP
      lwz     r4,16(r3)          ; descriptor address
      lwz     r5,0(r4)           ; first image address
idbcl:
      subi    r5,4               ; adjust address
dbcl:
      lwzu    r2,4(r4)           ; get flag word
      cmpi    r2,0               ; test flag
      beq     start              ; end continue
      andis.  r2,$6000           ; test for moveable code segment
      beq     skip               ; yes, skip it
      lwzu    r6,4(r4)           ; ram start address
      lwzu    r2,4(r4)           ; code end address
      subi    r2,4               ; adjust address
      sub     r2,r5              ; block size
      srwi    r2,2               ; word size
      mtctr   r2
```

```
	subi	r6,4			; adjust address
cbcl:
	lwzu	r0,4(r5)		; get and
	stwu	r0,4(r6)		; store
	bdnz	cbcl			; count down and loop back
	b	dbcl			; next segment
skip:
	lwzu	r5,8(r4)		; load next code address
	b	idbcl			; and loop back
start:
	lwz	r13,0(r3)		; initialize DP
	lwz	r2,20(r3)		; initialize R2 to point at constants
	bl	_main			; execute main
_exit:
	b	_exit			; stay here
;
	end
```

Let’s comment the most significant parts:

```
xdef		_exit, __stext
```

is needed to make sure that some symbols defined in this file are “visible” to other modules that need them. For example the __stext symbol is used in the BAM (you can check the source file bam.c that is provided with the Cosmic exmaples) in such a way that the execution jumps to this label just after the chip reset.

```
xref _main, __memory, __sdata, __sbss, __stack, __sconst, __idesc__
```

this line tells the assembler that these symbols are used in this file, but are defined elsewhere. Some of them are defined in the source files of the application (in this case only one: _main) and some are defined in the linker file. Note that, for those symbols defined in C source files, you need to add an underscore (“_”) at the beginning in order to use them in assembler.

```
ifdef VLE
	vle	on
.vtext:	section	.text
endif
```

this block will assemble the file in VLE or Standard mode depending on the fact that the VLE symbol is defined -> you must add a "-dVLE" on the assembler command line if you use a VLE derivative, otherwise your application will not work.

After this, there are two main loops: one that will initialize the whole memory to zero , and another that will initialize the "initialized data segments" (that is, all segment with the –id options: typically .sdata and/or .data, but you can add your own segments containing initialized data and they will be managed here)
Note: due to the RAM Error Correction Code (ECC) implemented in most PPC chips, it is mandatory that the whole RAM is set to 0 in a loop that write 4 bytes at a time before any other kind of RAM access (reading or writing 1 byte or 1 half word), as it is done in this example.

In the end, the registers are initialized:

```
init:
    lwz     r13,0(r3)       ; initialize DP
    lwz     r2,20(r3)       ; initialize R2 to point at constants
```

and the execution jumps to the application (that MUST start at "main()")

```
    bl      _main           ; execute main
```