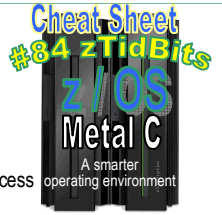


• In earlier z/OS Releases, all XL† C compiler-generated code required Language Environment.
- In addition to depending on the C runtime library functions that are available only with Language Environment, the generated code depended on the establishment of an overall execution context, including the heap storage and dynamic storage areas.
- These dependencies prohibit you from using the XL C compiler to generate code that runs in an environment where Language Environment did not exist.
• The XL C METAL compiler option, introduced in z/OS V1R9, generates code that does **not** require access to the Language Environment support at run time. Instead, the METAL option provides C-language extensions that allow you to specify assembly statements that call system services directly.
• Using these language extensions, you can provide almost any assembly macro, and your own function prologs and epilogs, to be embedded in the generated HLASM source file.
• When you understand how the METAL-generated code uses MVS™ linkage conventions to interact with HLASM code, you can use this capability to write freestanding programs.
- Because a freestanding program does not depend on any supplied runtime environment, it must obtain the system services that it needs by calling assembler services directly.

If you use the METAL compiler option together with XL C optimization capabilities, you can use C to write highly optimized system-level code.

Metal C environment Some functions require that an environment be created before they are called.
• You can create the environment by using a new function, __cinit() where this function will set up the appropriate control blocks and return an environment token to the caller.
- The caller must then ensure that GPR 12 contains this token when calling Metal C functions that require an environment.
NOTE: When the environment is no longer needed, a new function, __cterm(), can be used to perform cleanup, freeing all resources that had been obtained by using the token.
- An environment created by __cinit() can be used in both AMODE 31 and AMODE 64.
- In conjunction with this, the Metal C run time maintains both a below-the-bar heap and an above-the-bar heap for each environment.
> Calls to __malloc31() always affect the below-the-bar heap.
> Calls made in AMODE 31 to all other functions that obtain storage will affect below-the-bar heap.
> Calls made in AMODE 64 affect the above-the-bar heap.
• The storage key for all storage obtained on behalf of the environment is the psw key of the caller.
• The caller needs to ensure that the environment is always used with the same or compatible key.

The term **"Metal"** can denote **"raw"** and **"fundamental"**. Thus, the code generated by this option should be the fundamental code sequence that can run with minimal environmental dependencies

Because Metal C follows MVS linkage conventions, it enables the compiler-generated code to interoperate directly with the existing code base

The Metal C Runtime Library is a set of LPA-resident C functions that can be called from a C program created using the z/OS XL C compiler Metal option.

Programming with Metal C When you want to build an XL C program that can run in any z/OS environment, you can use the Metal C programming features provided by the XL C compiler as a high level language (HLL) alternative to writing the program in assembly language.
•Metal C programming features facilitate direct use of operating system services.
- Example, you can use the C programming language to write installation exits.
- When the METAL option is in effect, the XL C compiler generates code that is independent of Language Environment.
Note: Although the compiler generates default prolog and epilog code that allows the Metal C code to run, you might be required to supply your own prolog & epilog code to satisfy runtime environment requirements.
- Generates code that follows MVS linkage conventions and facilitates interoperations between the Metal C code and the existing code base.

The XL C compiler provides AR-mode programming support under the Metal option.

Important ☛ Metal C also provides a feature that improves the program's runtime performance.
☛ Provides support for accessing the data stored in data spaces (see: #54 zTidBits E x Addr.)
☛ Provides support for embedding your assembly statements into the compiler-generated code

• Metal C code needs to use dynamic storage area (DSA) as stack space.
- Each time a function is called, its prolog code acquires this space and, when control is returned to the calling function, its epilog code releases the stack space.
• Metal C avoids excessive acquisition and release operations by providing a mechanism that allows a called function to rely on pre-allocated stack space.
- This mechanism is the next available byte (NAB). All Metal C runtime library functions, as well as functions with a default prolog code, use it and expect the NAB address to be set by the calling function.
- The code that is generated to call a function includes the setup instructions to place the NAB address in the "address of next save area" field in the save area.
-The called function simply goes to the calling function's save area to pick up the NAB address that points to its own stack space.
- As a result, the called function does not need to explicitly obtain and free its own stack space.

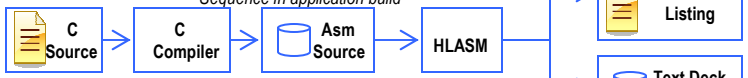
Metal C avoids excessive acquisition and release operations by providing a mechanism that allows a called function to rely on pre-allocated stack space. This mechanism is the next available byte (NAB). All Metal C runtime library functions, as well as functions with a default prolog code, use it and expect the NAB address to be set by the calling function.

† The IBM XL compiler family offers compilers built on an industry wide reputation for robustness, versatility and standards compliance. The true strength of the XL compilers comes through in its optimization and the ability to improve code generation. Optimized code executes with greater speed, using less machine resources, making you more productive.


Some metal pieces

Metal option overview **This** new mode of code generation was added to the XL C compiler.
• The Metal option serves as the switch to enable this new mode of code generation so that the code generated follows the standard MVS linkage conventions and does not have Language Environment dependencies.
• In the new mode of code generation, you can specify:
- The #pragma prolog/epilog directives that can be used to supply a programmer's own prolog and epilog code
- The __asm syntax that can be used to supply embedded HLASM statements
- The new language constructs and facilities to support accessing data stored in data spaces
- The subset of C library functions available for this mode
• With this new mode of code generation, it is therefore possible to use C language to:
- Write installation exits
- Develop programs that runs without Language Environment assistance
- Have a much simpler way of writing AR mode programs.
• By definition, a C function needs a stack space to store function scope variables and temporary storage for compiler-generated code.
NOTE: Normally, Language Environment supplies the stack space. When the generated code has no Language Environment at hand, the stack space may need to be supplied by the programmer.
• Without Language Environment, C library functions are unavailable, thus there needs to be a way for the C function to enlist system services directly and therefore, the code has to be compatible with the linkage expected by corresponding MVS system components.
• With the Metal option, other capabilities are enabled such as supplying a user's own prolog and epilog code, embedding a user's own HLASM statements in the C source code, and AR mode programming.

Sequence in application build



XL C programs are normally built by compiling C source files into object files that are linked into a program. When building Metal C programs, this process has an extra step: C source files are first compiled into HLASM source files and then assembled into object files, as shown above.

• The C-generated programs can now be mixed with programs written in HLASM.
• AMODE 64 code is also supported.
• z/Architecture hardware is required.

GENASM option To allow programmer-injected HLASM statements, the compiler needs to produce code in HLASM source code.
• The GENASM option can be used to name the output HLASM source file.
• The GENASM option also enables the compiler to process the __asm statements.
• The **–S** flag on the **xlc** command is equivalent to specifying the GENASM option.
- Currently, the GENASM option can only be used with the Metal option.

Programmer-supplied prolog and epilog code The compiler generates default prolog and epilog code which operates on a 1 M stack space.
• It may be necessary for the programmer to supply prolog and epilog code to set up whatever the environment
• There are several ways to supply customized prolog and epilog code:
- Using the #pragma prolog/epilog directives to apply the prolog/epilog to a single function
- Using the PROLOG and EPILOG options to apply the same prolog and epilog to all extern functions in the source file.
• Global set symbols are used to communicate compiler information to the user code; the intended use of this capability is for the programmer to specify a macro that contains the prolog/epilog code.

Programmer-embedded HLASM statements An example of a printf like __asm statement for embedding user HLASM statements using the following syntax is shown here:

__asm (code_format_string : output : input : clobbers)

Where ☛ code_format_string is a string literal similar to a printf format specifier.
☛ output is a comma-separated list of output operands; the list is optional.
☛ input is a comma separated list of input operands; the list is optional.
☛ clobbers is a comma-separated list of clobber registers; the list is optional.

NOTE: The colons separating output, input, and clobbers are required because these components are specified by position. Any or all of these components can be omitted.