```
    .maxstack 1
    ldstr "Hello world!"
    call void [mscorlib]System.Console::WriteLine(class System.String)
    ret
}
```

The **.assembly extern** declaration references an external assembly, mscorlib, which defines `System.Console`. The **.assembly** declaration in the second line declares the name of the assembly for this program. (Assemblies are the deployment unit for executable content for the CLI.) The **.method** declaration defines the global method `main`. The body of the method is enclosed in braces. The first line in the body indicates that this method is the entry point for the assembly (**.entrypoint**), and the second line in the body specifies that it requires at most one stack slot (**.maxstack**).

The method contains only three instructions. The **ldstr** instruction pushes the string constant `"Hello world!"` onto the stack, and the **call** instruction invokes `System.Console::WriteLine`, passing the string as its only argument (note that string literals in CIL are instances of the standard class `System.String`). As shown, call instructions shall include the full signature of the called method. Finally, the last instruction returns (**ret**) from `main`.

ANNOTATION: There is a common misunderstanding that **maxstack** refers to how much stack space is needed when the program runs. Instead, it is the maximum number of virtual machine slots needed in a particular method. When the method is converted to assembly, more or fewer actual machine words may be needed. A single virtual slot can hold an entire value type, which could be 100 words long—actual size is not a part of the virtual slot. Maxstack is there primarily for verifiers, and has no real effect on running the program. This information about virtual slots is needed when a verifier simulates a program, to tell the verifier how much space to allocate to represent that stack when it simulates this program.

4.2 Examples

This document contains integrated examples for most features of the CLI metadata. Many sections conclude with an example showing a typical use of the feature. All these examples are written using the ilasm assembly language. In addition, Partition V contains a longer example of a program written in the ilasm assembly language. All examples are, of course, informative only.

End informative text

1.7.2 *Valid Branch Targets*

The set of addresses composed of the first byte of each instruction identified in the instruction stream defines the only valid instruction targets. Instruction targets include branch targets as specified in branch instructions, targets specified in exception tables such as protected ranges (see Partition I, section 12.4.2 and Partition II, section 18), filters, and handler targets.

Branch instructions specify branch targets as either a 1-byte or a 4-byte signed relative offset; the size of the offset is differentiated by the opcode of the instruction. The offset is defined as being relative to the byte following the branch instruction. (**Note**: Thus, an offset value of zero targets the immediately following instruction.)

The value of a 1-byte offset is computed by interpreting that byte as a signed 8-bit integer. The value of a 4-byte offset can be computed by concatenating the bytes into a signed integer in the following manner: the byte of lowest address forms the least significant byte, and the byte with the highest address forms the most significant byte of the integer. (**Note**: This representation is often called "a signed integer in little-endian byte-order.")

ANNOTATION: The major point of this section is that it is invalid to jump into the middle of an instruction. This would be of concern in instructions that take more than 1 byte.

1.7.3 *Exception Ranges*

Exception tables describe ranges of instructions that are protected by catch, fault, or finally handlers (see Partition I, section 12.4.2 and Partition II, section 18). The starting address of a protected block, `filter` clause, or handler shall be a valid branch target as specified in Partition III, section 1.7.2. It is invalid for a protected block, `filter` clause, or handler to end without forming a complete last instruction.

ANNOTATION: The focus of this section is that multi-byte instructions cannot be split across exception boundaries.

1.7.4 *Must Provide Maxstack*

Every method specifies a maximum number of items that can be pushed onto the CIL evaluation [stack]. The value is stored in the `IMAGE_COR_ILMETHOD` structure that precedes the CIL body of each method. A method that specifies a maximum number of items less than the amount required by a static analysis of the method (using a traditional control

flow graph without analysis of the data) is invalid (hence also unverifiable) and need not be supported by a conforming implementation of the CLI.

NOTE

Maxstack is related to analysis of the program, not to the size of the stack at runtime. It does not specify the maximum size in bytes of a stack frame, but rather the number of items that must be tracked by an analysis tool.

RATIONALE

By analyzing the CIL stream for any method, it is easy to determine how many items will be pushed onto the CIL evaluation stack. However, specifying that maximum number ahead of time helps a CIL-to-native-code compiler (especially a simple one that does only a single pass through the CIL stream) in allocating internal data structures that model the stack and/or verification algorithm.

ANNOTATION: **Maxstack** is the maximum number of values pushed onto the evaluation stack within the method. It should not be confused with the space allocated at runtime. Maxstack is provided for JIT compilers and verifiers because they need to allocate space to simulate the method evaluation stack. Language compilers are required to compute this required space so that interpreters and JIT compilers can be written without having to scan the method before simulating or executing it.

1.7.5 *Backward Branch Constraints*

It must be possible, with a single forward-pass through the CIL instruction stream for any method, to infer the exact state of the evaluation stack at every instruction (where by "state" we mean the number and type of each item on the evaluation stack).

In particular, if that single-pass analysis arrives at an instruction, call it location X, that immediately follows an unconditional branch, and where X is not the target of an earlier branch instruction, then the state of the evaluation stack at X, clearly, cannot be derived from existing information. In this case, the CLI demands that the evaluation stack at X be empty.