Mandatory Eilision - NRVO by TPK Healy

Draft No. 1 2023-08-11

This paper describes a way of ensuring the mandatory eilision of a move/copy operation when returning a class by value from a function. Mandatory elision is guaranteed by the C++ Standard since C++17 only when returning a **prvalue**, for example with the following function:

```
#include <mutex>
using std::mutex;
mutex Func(void)
{
    return mutex();
}
```

Even though a '`mutex`' is unmovable and noncopyable, the above function compiles and runs fine because of *Return Value Optimisation* (RVO). The following function however won't compile:

```
mutex Func(void)
{
    mutex mtx;
    mtx.lock();
    return mtx;
}
```

It won't compile because with *Named Return Value Optimisation* (NRVO), the '`mutex`' class is required to have an accessible move/copy constructor, and so it is impossible to return a locked mutex by value from a function. This paper however describes a way of making this possible in four steps:

Step 1) Include the header file '`nrvo.hpp`' `#include "nrvo.hpp"`

Step 2) Change the signature of the function, making it return '`void`', and put a pointer to the return type in the place of the first parameter, as follows:

```
mutex Func(void)                →    void Func(mutex*)
mutex Func(int)                 →    void Func(mutex*,int)
mutex Func(double,string&&)     →    void Func(mutex*,double,string&&)
```

Step 3) Inside the body of the function, use '*placement* `new`' to construct the object:

```
#include <new>  // placement 'new'
void Func(mutex *const p, int const n)
{
    ::new(p) mutex();
    if ( 7 == n ) p->lock();   // but what if this line throws?
}
```

Step 4) Invoke the function using the template function '`nrvo`' as follows:

```
int main(void)
{
    auto mtx = nrvo(Func,7);
    mtx.unlock();
}
```

On Page 2 we will discuss what to do if an exception is thrown.

On the previous page, we had the following code for **Step 3**:

```
void Func(mutex *const p, int const n)
{
    ::new(p) mutex();
    if ( 7 == n ) p->lock();    // but what if this line throws?
}
```

There is a problem with the above function as the '`lock`' method might throw an '`std::system_error`'. If it throws, the destructor for the object will never be called. And so from the point that we construct the object, we must enclose the remainder of the function inside a try-catch block in order to ensure that the object gets destroyed before the exception propagates, as follows:

```
void Func(mutex *const p, int const n)
{
    ::new(p) mutex();
    try
    {
        if ( 7 == n ) p->lock();
    }
    catch(...) { p->~mutex(); throw; }
}
```

An alternative to a try-catch block is to use a helper class that will automatically destroy the object – but you must make sure to disarm the helper before returning from the function, as follows:

```
template<typename T>
class DestroyIfThrow {
    bool armed;
    T *const p;
public:
    void Disarm(void) noexcept { armed = false;  }
    explicit DestroyIfThrow(T *const arg) noexcept : armed(true), p(arg) {}
    ~DestroyIfThrow(void) { if ( armed ) p->~T(); }
};

void Func(mutex *const p, int const n)
{
    ::new(p) mutex();
    DestroyIfThrow dit(p);
    if ( 7 == n ) p->lock();
    dit.Disarm();
}
```

You don't need a try-catch block if you only invoke functions that are marked as '`noexcept`', for example the following is safe:

```
void Func(std::atomic_flag *const p)
{
    ::new(p) std::atomic_flag();
    p->test_and_set();              // this line won't throw
}
```

On Page 3 we discuss putting the object on the heap, in static duration memory, or inside an '`std::optional`'.

A standalone function named ‘PutRetvalIn’ is provided to put a function’s return value at an arbitrary memory address, such as on the heap in the following example:

```
#include <cstdlib>    // malloc, free
mutex Func(int, double, float)
{
    return mutex();
}
int main(void)
{
    mutex *const p = static_cast<mutex*>(  std::malloc(sizeof(mutex))  );
    PutRetvalIn(p)(Func)(5, 6.3, 8.4f);
    p->lock();
    p->unlock();
    p->~mutex();
    std::free(p);
}
```

The standalone function ‘PutRetvalIn’ can also be used to put the return value inside an ‘std::optional’ object as follows:

```
#include <optional>
mutex Func(int, double, float)
{
    return mutex();
}
std::optional<mutex> myglobal;
int main(void)
{
    PutRetvalIn(myglobal)(Func)(5, 6.3, 8.4f);
    myglobal->lock();
    myglobal->unlock();
}
```

Furthermore for added convenience, a function named ‘nrvo_PutRetvalIn’ is provided which is a combination of ‘nrvo’ and ‘PutRetvalIn’, which can be used as follows:

```
void Func(mutex const *p, int, double, float)
{
    ::new(p) mutex();
}
std::optional<mutex> myglobal;
int main(void)
{
    nrvo_PutRetvalIn(myglobal)(Func)(5, 6.3, 8.4f);
    myglobal->lock();
    myglobal->unlock();
}
```

The implementations of ‘nrvo’, ‘PutRetvalIn’ and “nrvo_PutRetvalIn’ are discussed on Page 4.

The implementations of '`nrvo`', '`PutRetvalIn`' and "`nrvo_PutRetvalIn`' are tested and working on the following operating systems, compilers and CPU instruction sets:

Operating System	*Compiler*	*Architecture*
Linux	GNU g++	alpha, arm32, **arm64**, **hppa**, **m68k** mips, mips64 mipsisa32r6, mipsisa64r6 powerpc32, powerpc64 riscv64, s390x, **sh4**, sparc64, x86_32, x86_64
MS-Windows	Microsoft Visual C++ Embarcadero (formerly Borland) GNU g++	x86_32, x86_64
Linux MS-Windows macOS	LLVM clang++	x86_64
Linux	Intel	x86_64

There are workflows setup on Github for the above 27 targets:

https://github.com/healytpk/nrvo/actions

Except for the four architectures highlighted in **blue** in the above table, all of the listed architectures and calling conventions have something in common: When a function returns a class by value, the caller function passes the address of allocated memory as the first argument to the callee function, and all the other arguments get moved down one position. So for example, on *Microsoft x64*, if a function would normally take two arguments in **rcx** and **rdx**, and if it returns a class by value, then **rcx** will be moved down to **rdx**, and **rdx** will be moved down to **r8**, while the pointer to the memory allocated for the return value will be put in **rcx**.

The four calling conventions in **blue** work differently though: **arm64**, **hppa**, **m68k** and **sh4**. On these four, the address of the return value is passed in a separate register, and so I needed to write a little helper function in assembler to move the address from the register into the position of the second parameter. This assembler function consists of three instructions:

(1) Copy the value of the 2nd parameter to a caller-saved scratch register
(2) Copy the address of the return value into the position of the 2nd parameter
(3) Jump to the address stored in the aforementioned caller-saved scratch register

Here are the four implementations written in assembler:

Motorola 6800 (m68k)	**Advanced Risc (ARM) 64-Bit (aarch64)**	**SuperH 4**	**Hewlett Packard Precision Architecture**

`move.l  8(%a7), %d0`
`move.l  %a1 , 8(%a7)`
`jmp  (%d0)`
	`mov  x9, x1`
	`mov  x1, x8`
	`br  x9`
		`mov  r5, r3`
		`mov  r2, r5`
		`jmp  @r3`
			`copy  %r25, %r20`
			`copy  %r28, %r25`
			`bv,n  %r0(%r20)`

The common implementation for all the other calling conventions is discussed on Page 5.

The common implementation of 'nrvo' does not require any assembler because of a trait common to all the other calling conventions. If you compile the following source file:

```
#include <new>                  // placement 'new'
#include <mutex>                // mutex
std::mutex Func1(void)
{
    return std::mutex();
}
void Func2(std::mutex *const p)
{
    ::new(p) std::mutex();
}
```

and then use 'objdump' to check the assembler produced by the C++ compiler, both functions will be identical – this is because the address of the return value is passed in the position of the first parameter. We take advantage of this trait in the common implementation of 'nrvo' as follows:

```
template<typename R, typename... Params>
R nrvo(std::function<void(R*,Params...)> const &f, Params... args)
{
    using F = std::function<void(R*,Params...)>;
    using FuncPtrT = void (*)(F const *,R*,Params...);

    auto const invoke_swap_1st_with_2nd =
    [](R *const r, F const *const f, FuncPtrT const funcptr, Params... args) -> void
    {
        // This helper function is needed in order to ensure that when calling
        // "std::function::operator()", the address of the "std::function" object
        // is placed in 1st position (instead of the return value address)
        funcptr( f, r, std::forward<Params>(args)... );
    };

    void (*const y)(R*, F const *, FuncPtrT, Params... args) = invoke_swap_1st_with_2nd;
    auto const z = reinterpret_cast<R (*)(F const *, FuncPtrT, Params...)>(y);

    auto const invoke_functor = [](F const *const p, R *const r, Params... args) -> void
    {
        // The GNU compiler g++ allows you to convert a member function pointer
        // to a normal function pointer:
        //    https://gcc.gnu.org/onlinedocs/gcc/Bound-member-functions.html
        // however some compilers such as LLVM clang++ don't -- and that's why
        // this helper function is needed.
        p->operator()(r, std::forward<Params>(args)... );
    };

    return z( &f, invoke_functor, std::forward<Params>(args)... );
}
```

The implementation of 'PutRetvalIn' is discussed on Page 6.

The common implementation of 'PutRetvalIn' invokes the following method:

```
void invoke(std::function<R(Params…)> const &f, Params... args)
{
    typedef std::function<R(Params…)> F;
    auto const invoke_functor = [](F const *const p, Params... args) -> R
        {
            return p->operator()( std::forward<Params>(args)... );
        };
    R (*const y)(F const *,Params...) = invoke_functor;
    auto const z = reinterpret_cast<void (*)(R*,F const*,Params...)>(y);
    z( static_cast<R*>(this->pricb.p), &f, std::forward<Params>(args)... );
}
```

The implementations of 'PutRetvalIn' for the aforementioned four architectures, **arm64**, **hppa**, **m68k** and **sh4** are a little more complicated. The 'invoke' method must not alter the stack nor alter any of the caller-saved registers nor the argument-passing registers, and yet somehow it must know the address of the 'std::function' object as well as the address of the function to jump to. I could have implemented this with the use of thread-local variables, but accessing threadlocal variables isn't straightforward in assembler (particularly on **arm64** as there are two separate ways of implementing threadlocal variables). I decided to use a technique whereby I would place the two pieces of data on the stack beside a known UUID, and so then the 'invoke' method would walk the stack backwards until it finds the UUID – and right beside it are the two pieces of data. The lines highlighted in **red** are where I put the UUID and two pieces of data on the stack:

```
void invoke(F const &f, Params... args)
{
    auto const invoke_functor = [](F const *const p, Params... args) -> R
      { return p->operator()( std::forward<Params>(args)... ); };

    R (*const y)(F const *, Params... args) = invoke_functor;
    auto const z = reinterpret_cast<void (*)(F const *, Params... args)>(y);

    static char unsigned const static_uuid[16u] = {
      0x20,0xb6,0x90,0x93,0xa4,0xa1,0x4e,0x79,0x93U,0xaf,0x36,0xdd,0xd0,0xe0,0xa3,0xc7 };

    alignas(16) char unsigned volatile
        stack_uuid[ 16u + sizeof(void*) + sizeof(void(*)(void)) ];

    char unsigned volatile *p = stack_uuid;

    for ( unsigned i = 0u; i < 16u; ++i ) *p++ = static_uuid[i];
    for ( unsigned i = 0u; i < sizeof(char unsigned*); ++i )
        *p++ = ((char unsigned const *)&pricb.p)[i];
    for ( unsigned i = 0u; i < sizeof(void(*)(void)); ++i )
        *p++ = ((char unsigned const *)&z)[i];

    auto const y = reinterpret_cast<void (*)(F const *,Params...)>(&invoke2);
    y( &f, std::forward<Params>(args)... );
}
```

The last line invokes 'y' which is the assembler function that walks the stack to find the UUID.

The full header file, '`nrvo.hpp`', is appended to this paper on Page 8, and can also be downloaded from:
http://virjacode.com/downloads/nrvo/nrvo.hpp

Links

There is a Github repository for '`nrvo`':
https://github.com/healytpk/nrvo/tree/main

The header file '`nrvo.hpp`' can be viewed in the Github repository at:
https://github.com/healytpk/nrvo/blob/main/nrvo.hpp

A test program '`nrvo_test.cpp`' can be viewed in the Github repository at:
https://github.com/healytpk/nrvo/blob/main/nrvo_test.cpp

The 27 workflows can be viewed in the Github repository at:
https://github.com/healytpk/nrvo/actions

Article about move/copy elision on the **cppreference.com** website:
https://en.cppreference.com/w/cpp/language/copy_elision

Discussion

Please respond to this paper on the C++ Standard Proposals Mailing List:

https://lists.isocpp.org/mailman/listinfo.cgi/std-proposals

You can view the mailing list archive here:

https://lists.isocpp.org/std-proposals/2023/08/date.php

Related papers:

- P2025R0 Guaranteed copy elision for named return objects by Anton Zhilin:
https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2025r0.html

---

You can reach the author of this paper, TPK Healy on t6@vir6jacode6.com (remove the 6's)

The full contents of the standalone header file 'nrvo.hpp' follow on Page 8.

```
001: #ifndef HEADER_INCLUSION_GUARD_NRVO_HPP
002: #define HEADER_INCLUSION_GUARD_NRVO_HPP
003:
004: // There are five implementations in this file:
005: // (1) The common implementation that works on most machines
006: // (2) m68k  : Motorola M68000
007: // (3) arm64 : ARM 64-Bit (aarch64)
008: // (4) sh4   : SuperH 4 (sh4)
009: // (5) hppa  : Hewlett Packard Precision Architecture
010:
011: #if !defined(__MC68K__) && !defined(__MC68000__) && !defined(__M68K__) && !defined(__m68k__) \
012:  && !defined(__aarch64__) && !defined(_M_ARM64)                                              \
013:  && !defined(__sh__) && !defined(__SH__) && !defined(__sh4__) && !defined(__SH4__)          \
014:  && !defined(__hppa__) && !defined(__hppa64__)                                             \
015: /* Don't remove this empty line -- it's needed */
016: #    define NRVO_USE_COMMON_IMPLEMENTATION
017: #endif
018:
019: #ifdef NRVO_USE_COMMON_IMPLEMENTATION  // ============ start of common code for 'nrvo' ============
020:
021: #include <cassert>           // assert
022: #include <functional>        // function
023: #include <optional>          // optional
024: #include <utility>           // move, forward
025: #include <variant>           // variant
026:
027: template<typename R, typename... Params>
028: R nrvo(std::function<void(R*,Params...)> const &f, Params... args)
029: {
030:     using F = std::function<void(R*,Params...)>;
031:     using FuncPtrT = void (*)(F const *,R*,Params...);
032:
033:     auto const invoke_swap_1st_with_2nd =
034:       [](R *const r, F const *const pf, FuncPtrT const fptr, Params... args2) -> void
035:       {
036:         // This helper function is needed in order to ensure that when calling
037:         // "std::function::operator()", the address of the "std::function" object
038:         // is placed in 1st position (instead of the return value address)
039:         fptr( pf, r, std::forward<Params>(args2)... );
040:       };
041:
042:     void (*const y)(R*,F const*,FuncPtrT,Params...) = invoke_swap_1st_with_2nd;
043:     auto const z = reinterpret_cast<R (*)(F const *,FuncPtrT,Params...)>(y);
044:
045:     auto const invoke_functor =
046:       [](F const *const p,R *const r,Params... args2) -> void
047:       {
048:         // The GNU compiler g++ allows you to convert a member function pointer
049:         // to a normal function pointer:
050:         //    https://gcc.gnu.org/onlinedocs/gcc/Bound-member-functions.html
051:         // however some compilers such as LLVM clang++ don't -- and that's why
052:         // this helper function is needed.
053:         p->operator()( r, std::forward<Params>(args2)... );
054:       };
055:
056:      return z( &f, invoke_functor, std::forward<Params>(args)... );
057: }
058:
059: #else  // ====================== end of common code for 'nrvo' ================================
060:
061: __asm__(
062: ".text                      \n"
063: "detail_nrvo_asm:           \n"
064: #  if defined(__MC68K__) || defined(__MC68000__) || defined(__M68K__) || defined(__m68k__)
065: "    move.l  8(%a7),   %d0  \n"                    // scratch_register = second_argument
066: "    move.l    %a1 , 8(%a7) \n"                    // second_argument  = address_of_return_value
067: "    jmp     (%d0)          \n"                    // program_counter  = scratch_register
068: #        define compare(value)                                \
069:             "addq.l    #1, %a0                          \n" \
070:             "move.b (%a0), %d1                          \n" \
071:             "move.l #0x" #value ", %d3                  \n" \
072:             "cmp.b    %d1, %d3                          \n" \
073:             "bne detail_nrvo_invoke2_loop               \n"
074: #  elif defined(__aarch64__) || defined(_M_ARM64)
```

```
075: "    mov     x9, x1          \n"                    // scratch_register = second_argument
076: "    mov     x1, x8          \n"                    // second_argument  = address_of_return_value
077: "    br      x9              \n"                    // program_counter  = scratch_register
078: #        define compare(value)                                  \
079:             "add x9, x9, #1                              \n" \
080:             "ldrb w10, [x9]                              \n" \
081:             "mov w11, #0x" #value "                      \n" \
082:             "cmp w10, w11                                \n" \
083:             "b.ne detail_nrvo_invoke2_loop               \n"
084: #  elif defined(__sh__) || defined(__SH__) || defined(__sh4__) || defined(__SH4__)
085: "    mov     r5, r3          \n"                    // scratch_register = second_argument
086: "    mov     r2, r5          \n"                    // second_argument  = address_of_return_value
087: "    jmp     @r3             \n"                    // program_counter  = scratch_register
088: "    nop                     \n"                    // extra instruction needed for alignment
089: #        define compare(value)                                  \
090:             "add #1, r0                                  \n" \
091:             "mov.b @r0, r1                               \n" \
092:             "mov #0x" #value ", r3                       \n" \
093:             "cmp/eq r1, r3                               \n" \
094:             "bf detail_nrvo_invoke2_loop                 \n"
095: #  elif defined(__hppa__) || defined(__hppa64__)
096: "    copy    %r25, %r20      \n"                    // scratch_register = second_argument
097: "    copy    %r28, %r25      \n"                    // second_argument  = address_of_return_value
098: "    bv,n    %r0(%r20)       \n"                    // program_counter  = scratch_register
099: #        define compare(value)                                    \
100:             "ldi 0x" #value ", %r20                       \n" \
101:             "ldb -1(%r21), %r22                           \n" \
102:             "cmpb,<> %r20, %r22, detail_nrvo_invoke2_loop \n" \
103:             "ldo -1(%r21), %r21                           \n"
104: #endif
105: );
106:
107: #define compare_bytes \
108:         compare(20)   \
109:         compare(b6)   \
110:         compare(90)   \
111:         compare(93)   \
112:         compare(a4)   \
113:         compare(a1)   \
114:         compare(4e)   \
115:         compare(79)   \
116:         compare(93)   \
117:         compare(af)   \
118:         compare(36)   \
119:         compare(dd)   \
120:         compare(d0)   \
121:         compare(e0)   \
122:         compare(a3)   \
123:         compare(c7)
124:
125: #include <cassert>           // assert
126: #include <climits>           // CHAR_BIT
127: #include <functional>        // function
128: #include <optional>          // optional
129: #include <utility>           // move, forward
130: #include <variant>           // variant
131:
132: extern "C" void detail_nrvo_asm(void);
133:
134: template<typename R, typename... Params>
135: R nrvo(std::function<void(R*,Params...)> const &f, Params... args)
136: {
137:     using F = std::function<void(R*,Params...)>;
138:
139:     auto const invoke_functor = [](F const *const p, R *const r, Params... args2) -> void
140:       {
141:         // The GNU compiler g++ allows you to convert a member function pointer
142:         // to a normal function pointer:
143:         //    https://gcc.gnu.org/onlinedocs/gcc/Bound-member-functions.html
144:         // however some compilers such as LLVM clang++ don't -- and that's why
145:         // this helper function is needed.
146:         p->operator()( r, std::forward<Params>(args2)... );
147:       };
148:
```

```
149:     void (*const funcptr)(F const*,R*,Params...) = invoke_functor;
150:
151:     auto const z = reinterpret_cast<R(*)(F const*,R*,Params...)>( &detail_nrvo_asm );
152:
153:     return z( &f, reinterpret_cast<R*>(funcptr), std::forward<Params>(args)... );
154: }
155:
156: #endif  // ifdef NRVO_USE_COMMON_IMPLEMENTATION
157:
158: // The following template allows for 'nrvo' to be used
159: // with a simple function pointer e.g. "void (*)(R*)"
160: // instead of an std::function object.
161: template<typename R, typename... Params>
162: R nrvo(void (*const funcptr)(R*,Params...), Params... args)
163: {
164:     return nrvo( std::function<void(R*,Params...)>(funcptr), std::forward<Params>(args)... );
165: }
166:
167: #if defined(__MC68K__) || defined(__MC68000__) || defined(__M68K__) || defined(__m68k__)
168:
169: __asm__(
170:     "detail_nrvo_invoke2:         \n"
171:         "move.l %a0, -(%a7)       \n"
172:         "move.l %d1, -(%a7)       \n"
173:         "move.l %d2, -(%a7)       \n"
174:         "move.l %d3, -(%a7)       \n"
175:         "move.l %a7,   %a0        \n" // Copy the stack pointer to %a0
176:         "subq.l  #1,   %a0        \n"
177:     "detail_nrvo_invoke2_loop:    \n"
178:          compare_bytes
179:         "addq.l    #1, %a0        \n"
180:         "move.l (%a0), %a1        \n" // The address of the return value
181:         "addq.l    #4, %a0        \n"
182:         "move.l (%a0), %d0        \n" // The address to jump to
183:         "move.l (%a7)+, %d3       \n"
184:         "move.l (%a7)+, %d2       \n"
185:         "move.l (%a7)+, %d1       \n"
186:         "move.l (%a7)+, %a0       \n"
187:         "jmp (%d0)                \n" // Jump to the address in %d3
188: );
189:
190: #elif defined(__aarch64__) || defined(_M_ARM64)
191:
192: __asm__(
193:     "detail_nrvo_invoke2:         \n"
194:         "mov x9, sp               \n"  // Copy the stack pointer to x9
195:         "sub x9, x9, #1           \n"  // Decrement x9 by 1 byte
196:     "detail_nrvo_invoke2_loop:    \n"
197:          compare_bytes
198:         "add x9, x9, #1           \n"
199:         "ldr x8, [x9]             \n"  // The address of the return value
200:         "add x9, x9, #8           \n"
201:         "ldr x9, [x9]             \n"  // The address to jump to
202:         "br x9                    \n");
203:
204: #elif defined(__sh__) || defined(__SH__) || defined(__sh4__) || defined(__SH4__)
205:
206: __asm__(
207:     "detail_nrvo_invoke2:         \n"
208:         "mov r15, r0              \n"
209:         "add #-1, r0              \n"
210:     "detail_nrvo_invoke2_loop:    \n"
211:          compare_bytes
212:         "add #1, r0               \n"
213:         "mov.l @r0, r2            \n"  // The address of the return value
214:         "add #4, r0               \n"
215:         "mov.l @r0, r3            \n"  // The address to jump to
216:         "jmp @r3                  \n");
217:
218: #elif defined(__hppa__) || defined(__hppa64__)
219:
220: // Stack grows upwards on HPPA -- every other architecture grows downards!
221: // That's why we search for the UUID backwards
222:
```

```
223: __asm__ (
224:     ".text                       \n"
225:     "detail_nrvo_invoke2:        \n"
226:         "copy %sp, %r21          \n"  // p = std::stack_pointer() (points at empty space)
227:         "ldo  1(%r21), %r21      \n"  // ++p    [20 b6 90 93 a4 a1 4e 79 93 af 36 dd d0 e0 a3 c7]
228:     "detail_nrvo_invoke2_loop:   \n"
229:         "ldo -1(%r21), %r21      \n"  // --p
230:         compare(c7)  // [20 b6 90 93 a4 a1 4e 79 93 af 36 dd d0 e0 a3 c7]
231:         compare(a3)
232:         compare(e0)
233:         compare(d0)
234:         compare(dd)
235:         compare(36)
236:         compare(af)
237:         compare(93)
238:         compare(79)
239:         compare(4e)
240:         compare(a1)
241:         compare(a4)
242:         compare(93)
243:         compare(90)
244:         compare(b6)
245:         compare(20)
246:         "ldo 0x10(%r21), %r21    \n"  // p += 16u
247:         "ldw 0(%r21), %r28       \n"  // address of retval = *(uint32_t*)r21
248:         "ldo 0x04(%r21), %r21    \n"  // p +=  4u
249:         "ldw 0(%r21), %r20       \n"  // void (*f)(void) = *(uint32_t*)r21
250:         "bv,n %r0(%r20)          \n");// f()
251:
252: #endif
253:
254: extern "C" void detail_nrvo_invoke2(void);
255:
256: namespace detail_nrvo {
257:
258:     template<typename R, typename... Params> class Invoker;
259:
260:     class PutRetvalIn_Class_Base {
261:         std::function<void(void)> cleanup;
262:         void *const p;
263:     public:
264:         PutRetvalIn_Class_Base(void *const arg_p, std::function<void(void)> &&arg_cleanup) noexcept
265:           : cleanup(std::move(arg_cleanup)), p(arg_p) {}
266:         PutRetvalIn_Class_Base(PutRetvalIn_Class_Base       &&) = delete;
267:         PutRetvalIn_Class_Base(PutRetvalIn_Class_Base const & ) = delete;
268:         PutRetvalIn_Class_Base &operator=(PutRetvalIn_Class_Base       &&) = delete;
269:         PutRetvalIn_Class_Base &operator=(PutRetvalIn_Class_Base const & ) = delete;
270:         template<typename R, typename... Params> friend class Invoker;
271:     };
272:
273:     template<typename R, typename... Params>
274:     class Invoker {
275:
276:         typedef std::function<R(Params...)> F;
277:
278:         // The constructor of this class is private so that you can't just pull
279:         // one out of thin air. There is only one friend: PutRetvalIn<R>
280:
281:         // An object of type 'Invoker<R,Params...>' is returned
282:         // by value from PutRetvalIn<R>::operator()<Params...>
283:
284:         PutRetvalIn_Class_Base &&pricb;
285:         std::variant<F const *, R(*)(Params...)> ptr;
286:
287:         Invoker(PutRetvalIn_Class_Base &&argP,       F const &argF        ) noexcept
288:             : pricb( std::move(argP) ), ptr(&argF) {}
289:
290:         Invoker(PutRetvalIn_Class_Base &&argP, R (*const argF)(Params...) ) noexcept
291:             : pricb( std::move(argP) ), ptr( argF) {}
292:
293:         void invoke(F const &f, Params... args)
294:         {
295:             auto const invoke_functor =
296:               [](F const *const p, Params... args2) -> R
```

```
297:               {
298:                 // The GNU compiler g++ allows you to convert a member function pointer
299:                 // to a normal function pointer:
300:                 //    https://gcc.gnu.org/onlinedocs/gcc/Bound-member-functions.html
301:                 // however some compilers such as LLVM clang++ don't -- and that's why
302:                 // this helper function is needed.
303:                 return p->operator()( std::forward<Params>(args2)... );
304:               };
305:
306:             R (*const y)(F const*,Params...) = invoke_functor;
307:
308:             try
309:             {
310: #ifdef NRVO_USE_COMMON_IMPLEMENTATION
311:                 auto const z = reinterpret_cast<void (*)(R*,F const*,Params...)>(y);
312:                 z( static_cast<R*>(pricb.p), &f, std::forward<Params>(args)... );
313: #else
314:                 auto const z = reinterpret_cast<void (*)(F const*,Params...)>(y);
315:
316:                 static_assert( 8u == CHAR_BIT, "Not ready yet for Texas Instruments microcontrollers");
317:
318:                 static char unsigned const static_uuid[16u] = {
319:                     0x20U, 0xb6U, 0x90U, 0x93U, 0xa4U, 0xa1U, 0x4eU, 0x79U,
320:                     0x93U, 0xafU, 0x36U, 0xddU, 0xd0U, 0xe0U, 0xa3U, 0xc7U,
321:                 };
322:
323:                 // This following 'stack_uuid' is where we're putting the UUID + data on the stack
324:                 alignas(16u) char unsigned volatile stack_uuid[16u +sizeof(void*) + sizeof(void(*)(void))
];
325:
326:                 char unsigned volatile *p = stack_uuid;
327:
328:                 for ( unsigned i = 0u; i < 16u; ++i )
329:                     *p++ = static_uuid[i];
330:                 for ( unsigned i = 0u; i < sizeof(void*); ++i )
331:                     *p++ = static_cast<char unsigned const*>(static_cast<void const*>(&pricb.p))[i];
332:                 for ( unsigned i = 0u; i < sizeof(void(*)(void)); ++i )
333:                     *p++ = reinterpret_cast<char unsigned const*>(&z)[i];
334:
335:                 auto const zz = reinterpret_cast<void (*)(F const*,Params...)>(&detail_nrvo_invoke2);
336:
337:                 zz( &f, std::forward<Params>(args)... );
338: #endif
339:             }
340:             catch (...)
341:             {
342:                 if ( pricb.cleanup ) pricb.cleanup();
343:
344:                 throw;
345:             }
346:         }
347:
348:     public:
349:         void operator()(Params... args) /* deliberately non-const */
350:         {
351:             if ( 0u == this->ptr.index() )
352:                 this->invoke(   *std::get<0u>(this->ptr)  , std::forward<Params>(args)... );
353:             else
354:                 this->invoke( F( std::get<1u>(this->ptr) ), std::forward<Params>(args)... );
355:         }
356:
357:         Invoker(Invoker       &&) = delete;
358:         Invoker(Invoker const & ) = delete;
359:         Invoker &operator=(Invoker       &&) = delete;
360:         Invoker &operator=(Invoker const & ) = delete;
361:
362:         friend class PutRetvalIn_Class;
363:         friend class PutRetvalIn_Class_nrvo;
364:     };
365:
366:     class PutRetvalIn_Class : public PutRetvalIn_Class_Base {
367:     public:
368:         explicit PutRetvalIn_Class(void *const arg_p, std::function<void(void)> &&arg_cleanup = {})
noexcept
```

```
369:           : PutRetvalIn_Class_Base(arg_p, std::move(arg_cleanup) ) {}
370:
371:         template<typename R, typename... Params>
372:         detail_nrvo::Invoker<R,Params...> operator()(std::function<R(Params...)> const &arg)
373:         {
374:             return detail_nrvo::Invoker<R,Params...>( std::move(*this), arg);
375:         }
376:
377:         template<typename R, typename... Params>
378:         detail_nrvo::Invoker<R,Params...> operator()( R(*const arg)(Params...) )
379:         {
380:             return detail_nrvo::Invoker<R,Params...>( std::move(*this), arg);
381:         }
382:
383:         PutRetvalIn_Class(PutRetvalIn_Class       &&) = delete;
384:         PutRetvalIn_Class(PutRetvalIn_Class const & ) = delete;
385:         PutRetvalIn_Class &operator=(PutRetvalIn_Class       &&) = delete;
386:         PutRetvalIn_Class &operator=(PutRetvalIn_Class const & ) = delete;
387:     };
388:
389:     class PutRetvalIn_Class_nrvo : public PutRetvalIn_Class_Base {
390:         alignas(std::function<void(void)>) char unsigned buf[sizeof(std::function<void(void)>)];
391:         std::function<void(void)> destroy_buf;
392:
393:         template<typename R, typename... Params>
394:         detail_nrvo::Invoker<R,Params...> common(std::function<R(Params...)> &&arg)
395:         {
396:             typedef std::function<R(Params...)> F;
397:             static_assert( sizeof (std::function<void(void)>) == sizeof (F) );
398:             static_assert( alignof(std::function<void(void)>) == alignof(F) );
399:
400:             F *const pf = static_cast<F*>(static_cast<void*>(this->buf));
401:             ::new(pf) F( std::move(arg) );
402:             destroy_buf = [pf](){ pf->~F(); };
403:
404:             return detail_nrvo::Invoker<R,Params...>( std::move(*this), *pf );
405:         }
406:
407:     public:
408:
409:         ~PutRetvalIn_Class_nrvo(void)
410:         {
411:             if ( destroy_buf ) destroy_buf();
412:         }
413:
414:         // cppcheck-suppress uninitMemberVar
415:         explicit PutRetvalIn_Class_nrvo(void *const arg_p, std::function<void(void)> &&arg_cleanup = {})
416:          noexcept : PutRetvalIn_Class_Base(arg_p, std::move(arg_cleanup) ) {}
417:
418:         template<typename R, typename... Params>
419:         detail_nrvo::Invoker<R,Params...> operator()(std::function<void(R*,Params...)> const &arg)
420:         {
421:             auto mylambda = [&arg](Params... args)->R { return nrvo(arg, std::forward<Params>(args)...);
};
422:             return this->common<R,Params...>( std::function<R(Params...)>(mylambda) );
423:         }
424:
425:         template<typename R, typename... Params>
426:         detail_nrvo::Invoker<R,Params...> operator()( void(*const arg)(R*,Params...) )
427:         {
428:             auto mylambda = [arg](Params... args)->R { return nrvo(arg, std::forward<Params>(args)... );
};
429:             return this->common<R,Params...>( std::function<R(Params...)>(mylambda) );
430:         }
431:
432:         PutRetvalIn_Class_nrvo(PutRetvalIn_Class_nrvo       &&) = delete;
433:         PutRetvalIn_Class_nrvo(PutRetvalIn_Class_nrvo const & ) = delete;
434:         PutRetvalIn_Class_nrvo &operator=(PutRetvalIn_Class_nrvo       &&) = delete;
435:         PutRetvalIn_Class_nrvo &operator=(PutRetvalIn_Class_nrvo const & ) = delete;
436:     };
437:
438:     template<class R, class Helper>
439:     Helper PutRetvalIn_optional(std::optional<R> &arg)
440:     {
```

```
441:         arg.reset();
442:
443:         struct Dummy {
444:             alignas(R) char unsigned buf[sizeof(R)];  // deliberately not initialised
445:             Dummy(void) {}                            // cppcheck-suppress uninitMemberVar
446:             Dummy(Dummy const & ) = delete;
447:             Dummy(Dummy       &&) = delete;
448:         };
449:
450:         std::optional<Dummy> &b = *static_cast< std::optional<Dummy> * >(static_cast<void*>(&arg));
451:         static_assert( sizeof(arg) == sizeof(b) );
452:
453:         b.emplace();  // This sets the 'has_value' to true
454:
455:         assert( static_cast<void*>(&arg.value()) == static_cast<void*>(&b.value()) );
456:
457:         return Helper( static_cast<void*>(&arg.value()), [&b](void)->void { b.reset(); } );
458:     }
459: } // close namespace 'detail_nrvo'
460:
461: detail_nrvo::PutRetvalIn_Class PutRetvalIn(void *const arg)
462: {
463:     return detail_nrvo::PutRetvalIn_Class(arg);
464: }
465:
466: detail_nrvo::PutRetvalIn_Class_nrvo nrvo_PutRetvalIn(void *const arg)
467: {
468:     return detail_nrvo::PutRetvalIn_Class_nrvo(arg);
469: }
470:
471: template<class R>
472: detail_nrvo::PutRetvalIn_Class PutRetvalIn(std::optional<R> &arg)
473: {
474:     return detail_nrvo::PutRetvalIn_optional< R, detail_nrvo::PutRetvalIn_Class >(arg);
475: }
476:
477: template<class R>
478: detail_nrvo::PutRetvalIn_Class_nrvo nrvo_PutRetvalIn(std::optional<R> &arg)
479: {
480:     return detail_nrvo::PutRetvalIn_optional< R, detail_nrvo::PutRetvalIn_Class_nrvo >(arg);
481: }
482:
483: #endif  //  ifndef HEADER_INCLUSION_GUARD_NRVO_HPP
```

See summary and internet links on Page 7