Integrating
Lua
into the high-end automation framework
McRobot

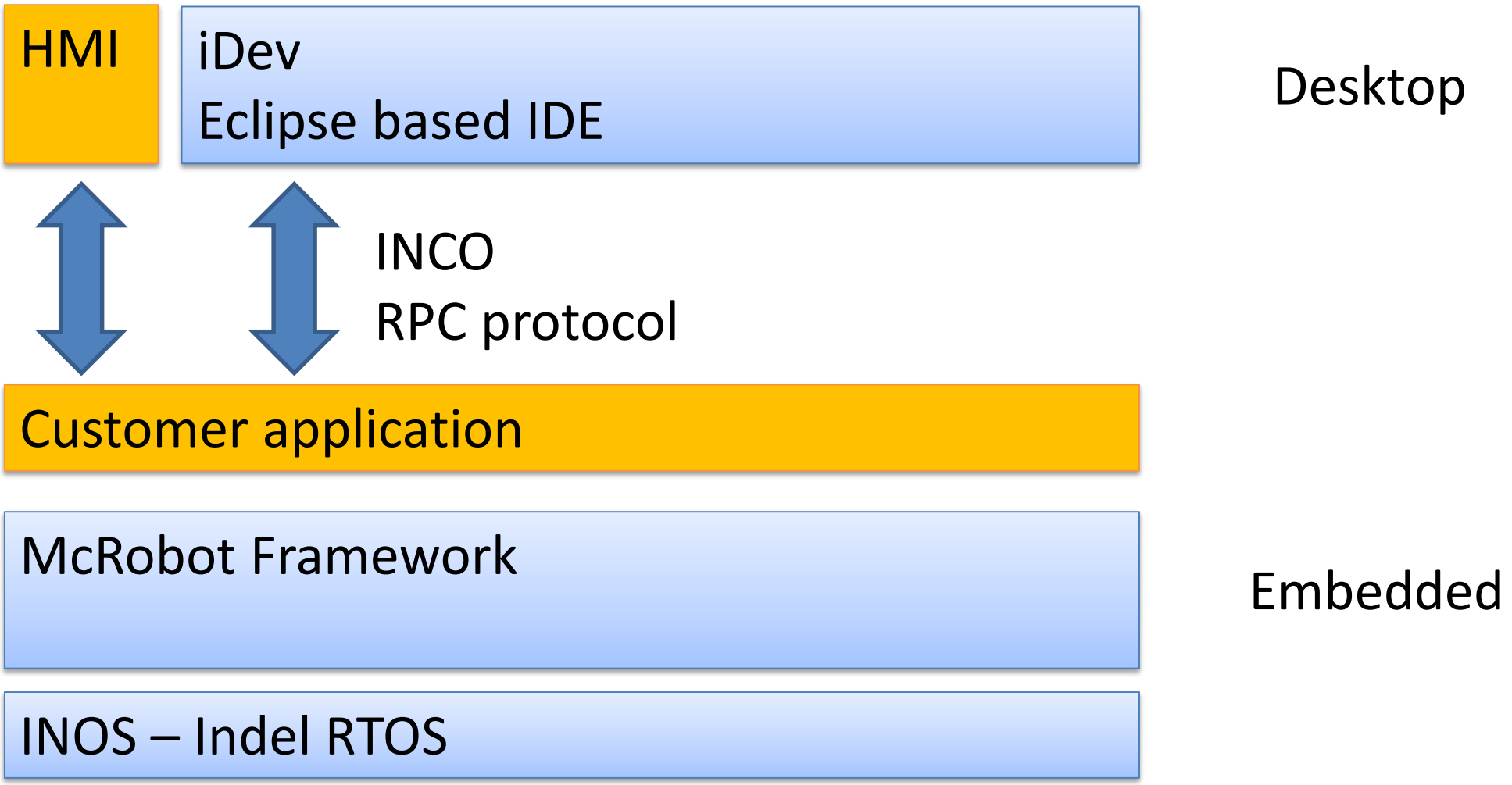
Specialized in high-end automation

33 Employees

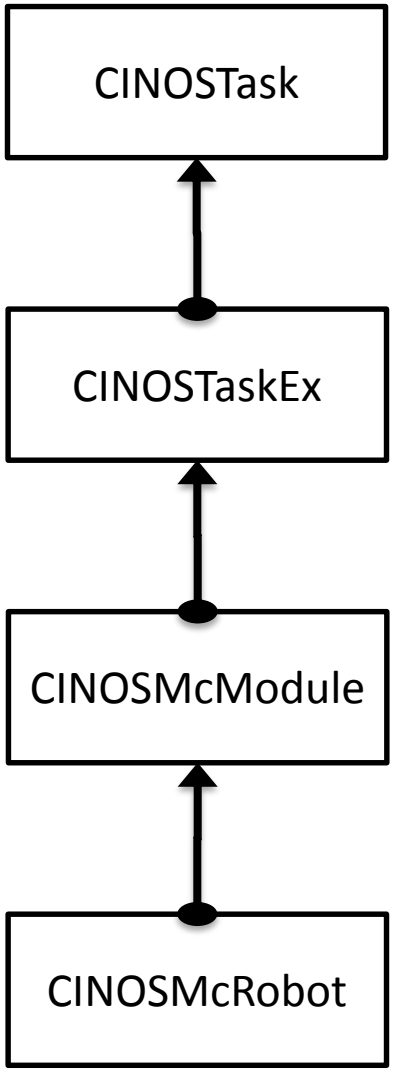
Founded in 1974

Based in Zürich, Switzerland

Machine control software



McRobot Framework



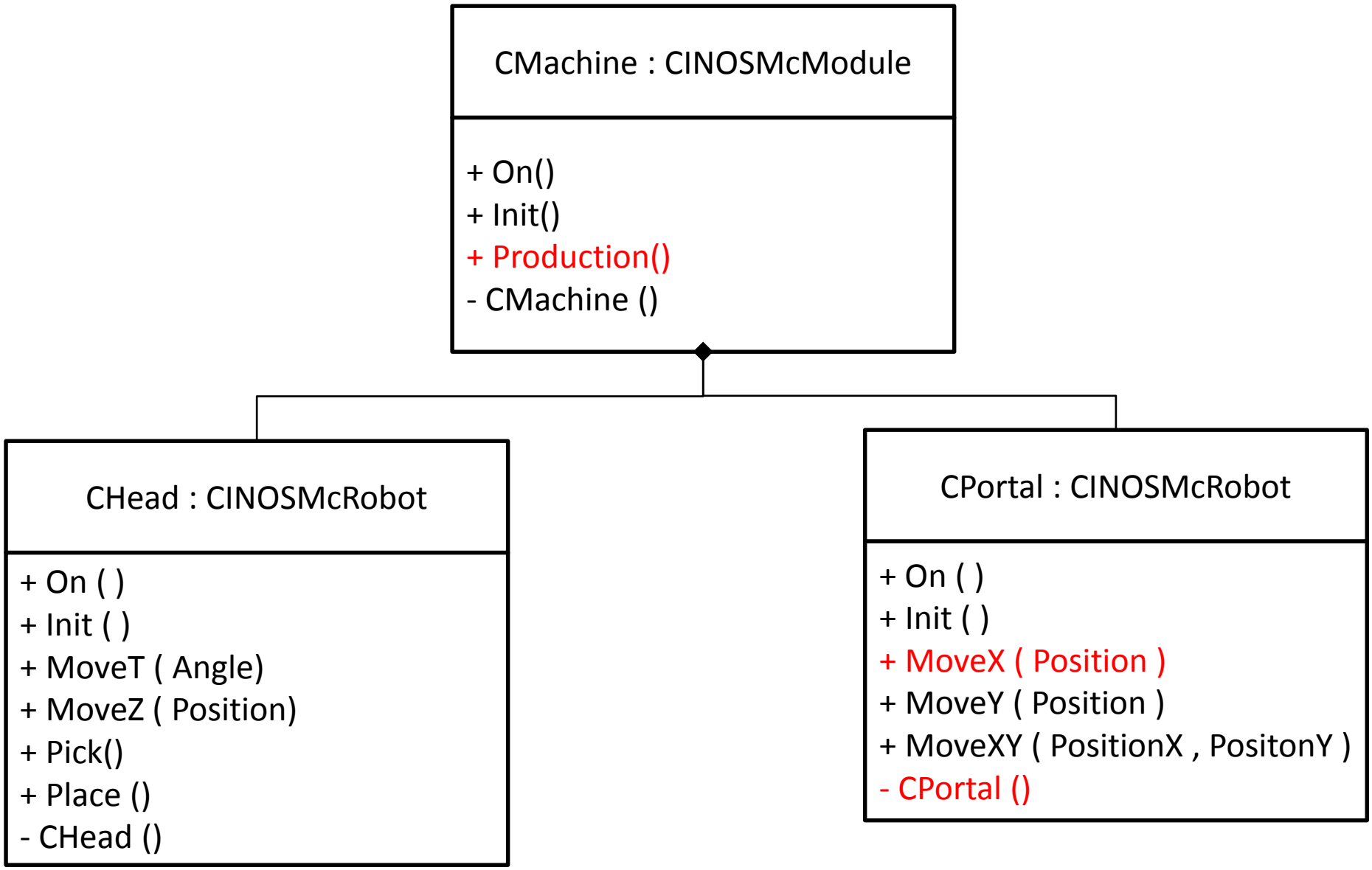
- Lightweight C++ task

- Message handling
- Commands

- Resources (I/O)
- Software representation of hardware modules

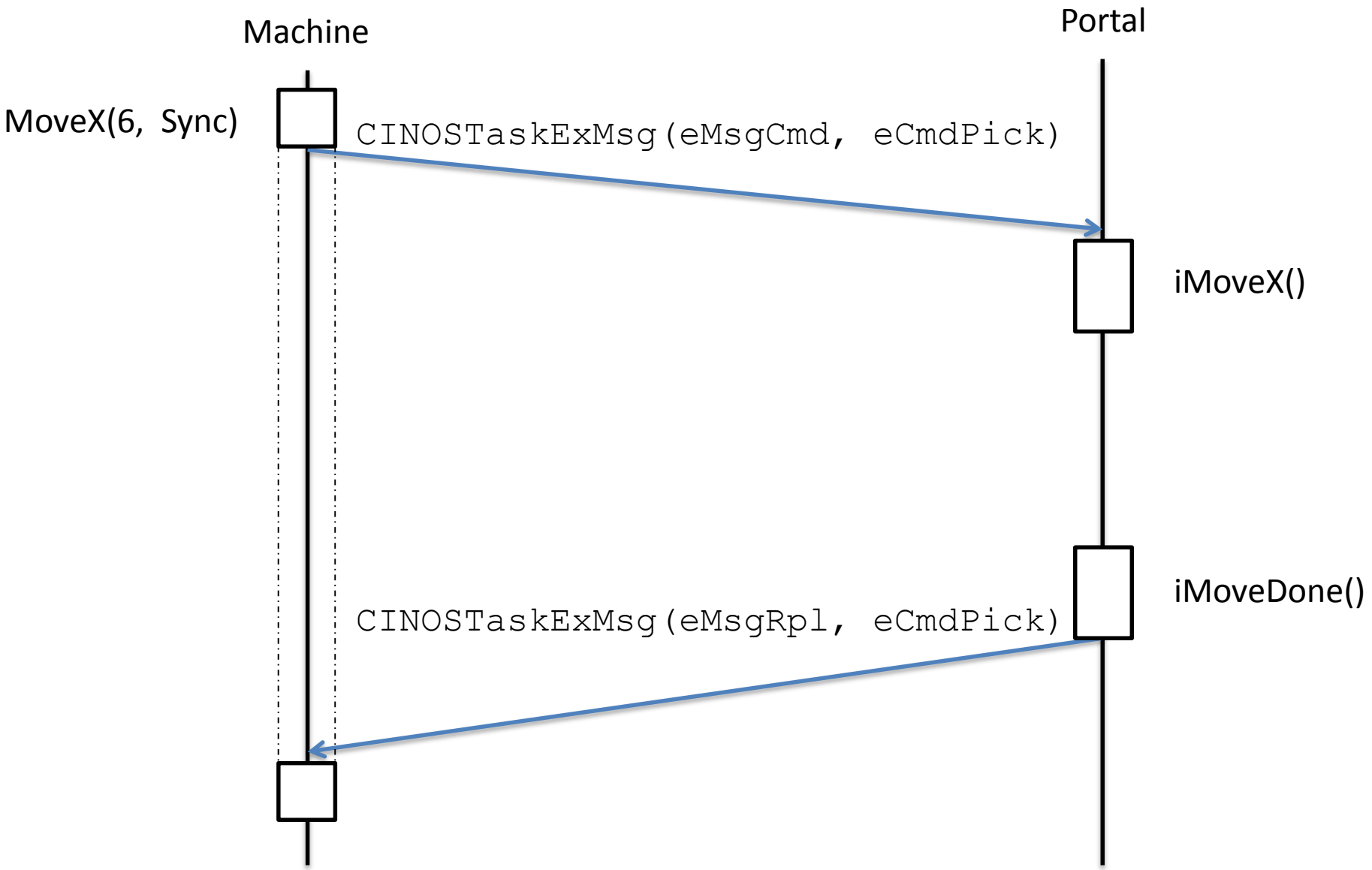
- Axes
- Moves

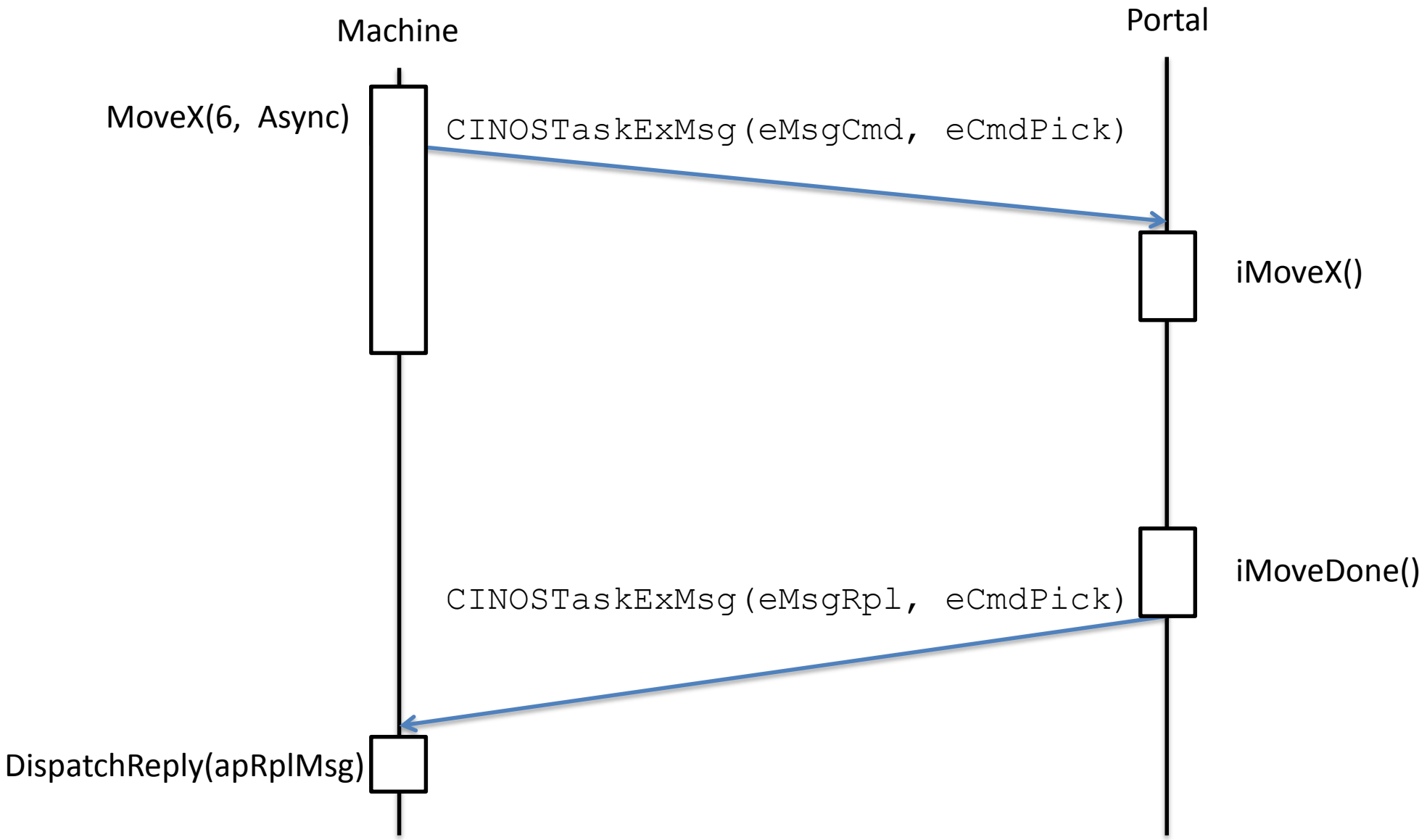
Pick and place machine



```
void CPortal::CPortal () {
        // Register MoveX, creates a prototype message object for MoveX
        AddCommand("MoveX", eCmdMoveX, MoveX);
        AddParam("Position", 0.0);

        // Register MoveY, creates a prototype message object for MoveY
        AddCommand("MoveY", eCmdMoveY, MoveY);
        AddParam("Position", 0.0);

        // Register MoveXY , creates a prototype message object for MoveXY
        AddCommand("MoveXY", eCmdMoveXY, MoveXY);
        AddParam("PositionX", 0.0);
        AddParam("PositionY", 0.0);
}
```

```
void CPortal::MoveX(double adPosition, CINOSSync* apSync) {
        // Create a message
        CINOSTaskExMsg  msg =
                        new CINOSTaskExMsg( eMsgCmd, eCmdMoveX, apSync);
        msg->AddParam(adPosition);
        // Put it in the message queue of this object
        PutMsg(msg);
}

void CPortal::iMoveX (CINOSTaskExMsg* apMsg) {
        double Position = apMsg->GetParam<double>();

        // Call move on axis, iMoveDone as callback
        m_pAxisX->Move(Position, iMoveDone );
        m_pMoveMsg = apMsg;
        MsgDefer(apMsg);
}

void CPortal::iMoveDone () {
        // Mark  as done, sends a reply message to sender task
        MsgDone(m_pMoveMsg);
}
```


Machine
Portal
MoveX(6, Sync)
CINOSTaskExMsg(eMsgCmd, eCmdPick)
iMoveX()
iMoveDone()
CINOSTaskExMsg(eMsgRpl, eCmdPick)


Machine
Portal
MoveX(6, Async)
CINOSTaskExMsg(eMsgCmd, eCmdPick)
iMoveX()
iMoveDone()
CINOSTaskExMsg(eMsgRpl, eCmdPick)
DispatchReply(apRplMsg)

```
void CMachine::iProduction(CINOSTaskExMsg* apMsg) {
        // Move Portal and head
        m_pPortal->MoveXY(100.0, 120.0, DEF_INOS_ASYNC);
        m_pHead->MoveT(30.0, DEF_INOS_ASYNC);
        // Wait until both are finish (5 seconds, maybe…)
        Sleep(5000);

        // Pick
        m_pHead->Pick(12.0, DEF_INOS_SYNC);

        // Move Portal and head
        m_pPortal->MoveXY(200.0, 10.0, DEF_INOS_ASYNC);
        m_pHead->MoveT(0.0, DEF_INOS_ASYNC);
        Sleep(5000);
        m_pHead->Place(4.0, DEF_INOS_SYNC);
        // ...
}
```

Is this good?

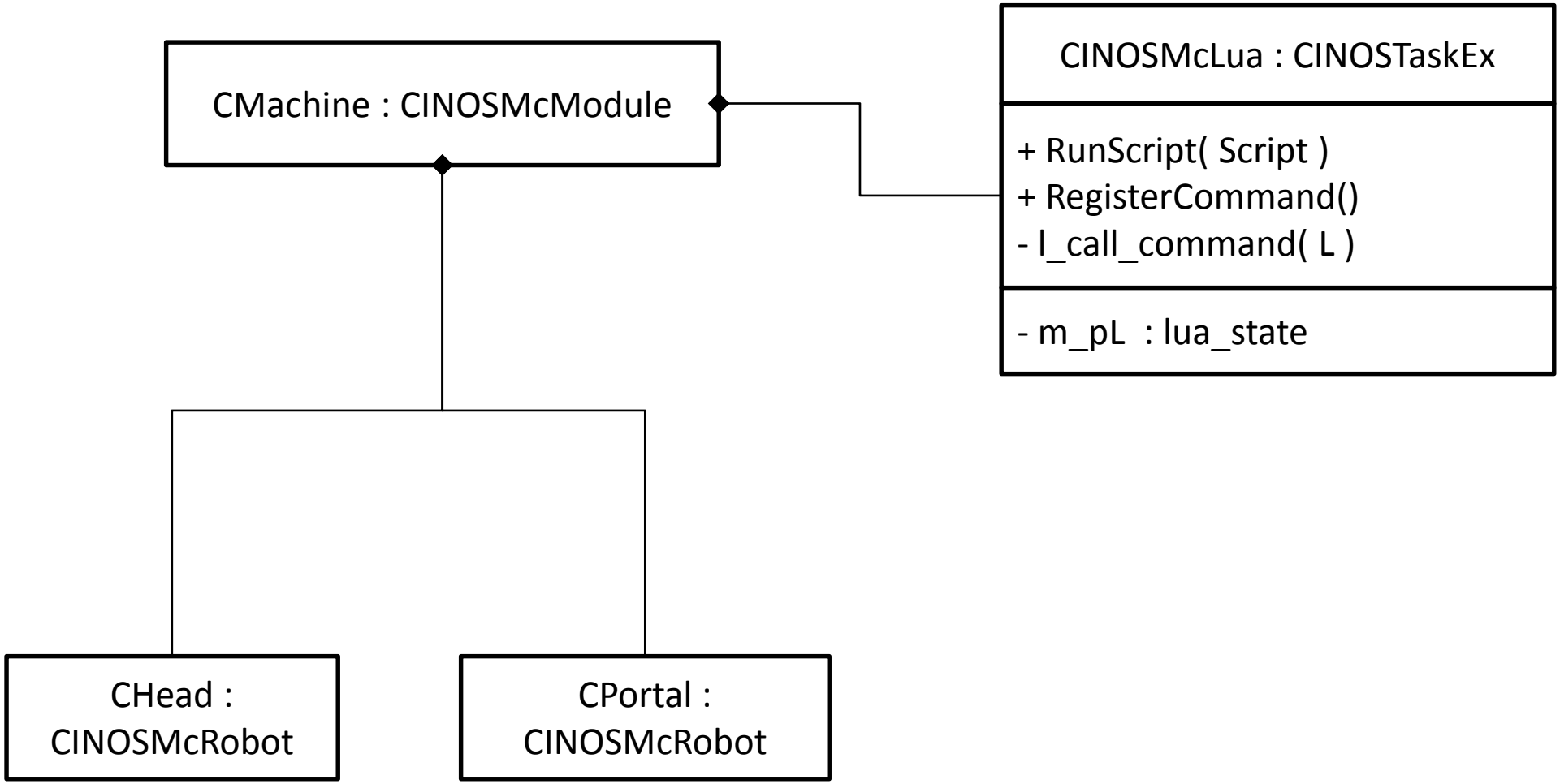
Binding Lua with McRobot

The goal

```
CMachine::iProduction(CINOSTaskExMsg* apMsg) {
        m_pPortal->MoveXY(100.0, 120.0, DEF_INOS_ASYNC);
        m_pHead->MoveT(30.0, DEF_INOS_ASYNC);
        Sleep(5000);
        m_pHead->Pick(12.0, DEF_INOS_SYNC);

        //..
}

function Production()
        Mc.Module.Portal.Cmd.MoveXY(100.0, 120.0, Async)
        Mc.Module.Head.Cmd.MoveT(30.0, Async)
        Sleep(5000)
        Mc.Module.Head.Cmd.Pick(12.0, Sync)

        -- ...
end
```

What do we need?

- Dedicated C++ Task for Lua interpreter
- Call Lua from McRobot
- Populate the global Lua table “Mc” with all McRobot commands
- Call McRobot commands from Lua


CMachine : CINOSMcModule
CINOSMcLua : CINOSTaskEx
+ RunScript(Script)
+ RegisterCommand()
- l_call_command(L)
- m_pL : lua_state
CHead : CINOSMcRobot
CPortal : CINOSMcRobot

```
void CINOSMcLua::iRunScript (CINOSTaskExMsg* apMsg) {
	const char* pScript = apMsg->GetParam<>();

	// load and parse the script
	if(luaL_loadstring(m_pL, pScript)) {
		MsgError(apMsg, INOS_MCMSG_CODE_LUA_COMPILE_ERROR);
		return;
	}
	// call it
	switch(lua_pcall(m_pL, 0, 0, 0)) {
		case LUA_OK:
			MsgDone(apMsg);
			return;

		case LUA_ERRRUN:
		case LUA_ERRERR:
			MsgError(apMsg, INOS_MCMSG_CODE_LUA_RUNTIME_ERROR);
			return;

		default: //shouldn't happen ☹
			ASSERT_ALWAYS(false);
	}
}
```

```
void CINOSMcLua::RegisterCommand (CINOSTaskExMsg* apMsg) {
        // Get the table with all modules
        lua_getglobal(m_pLuaMainState, "Mc");
        lua_getfield(m_pLuaMainState, -1, "Module");

        // Get the table of the right module (Mc.Module.Portal)
        lua_getfield(m_pLuaMainState, -1, apMsg->GetParam());

        // Get the table of the commands of the module (Mc.Module.Portal.Cmd)
        lua_getfield(m_pLuaMainState, -1, "Cmd");

        // Create a userdata of command message prototype
        CINOSTaskExMsg** pCommandData =
        (CINOSTaskExMsg**)lua_newuserdata(m_pLuaMainState, 4);
        *pCommandData = apMsg->GetParam<CINOSTaskExMsg*>();

        // Add the metatable "McModuleCmd"
        luaL_getmetatable(m_pLuaMainState, "McModuleCmd");
        lua_setmetatable(m_pLuaMainState, -2);

        // Userdata to table (Mc.Module.Portal.Cmd.MoveX)
        lua_setfield(m_pLuaMainState, -2, (*pCommandData)->GetName());
}
```

```
// __call of "McModuleCmd" metatable
int l_call_command(lua_State *L) {
	// Get the "this" pointer from ud
	CINOSMcLua* pMcLua = (CINOSMcLua*) lua_getud(L);
	// Get userdata (first argument of __call)
	CINOSTaskExMsg* pCmdMsg =
		*(CINOSTaskExMsg**)luaL_checkudata(L, 1, "McModuleCmd");
	 // Create a new message from prototype message
	CINOSTaskExMsg* pCallMsg =
		new CINOSTaskExMsg(CINOSTaskEx::eMsgCmd,
				pCmdMsg->GetMsgCode(),
				lua_tointeger(L, -1)); // Sync argument

	// Loop through the parameters
	CINOSTaskExMsgParam* pParam = pCmdMsg->GetFirstParamPtr();
	int index = 1;
	while(pParam != NULL) {
		AddParameter(L, pCallMsg, pParam, index++);
		pParam = pCmdMsg->GetParamPtr();
	}
	// Finally put the message
	pCmdMsg->GetTask()->PutMsg(pCallMsg, pCallMsg->GetId());
}
```

Custom Lua Scheduler

Or:

How

I Learned

To

Stop

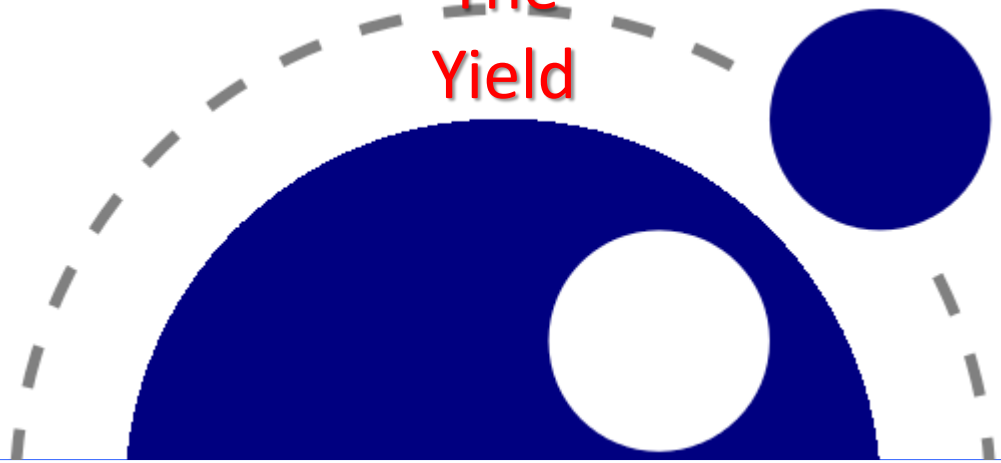
Worrying

And

Love

The

Yield



What do we want?

- Run more than one Lua script in parallel in the same C++ task.
- Synchronize with asynchonously started McRobot commands – WaitAll()
- Interupt Lua scripts (preemption)

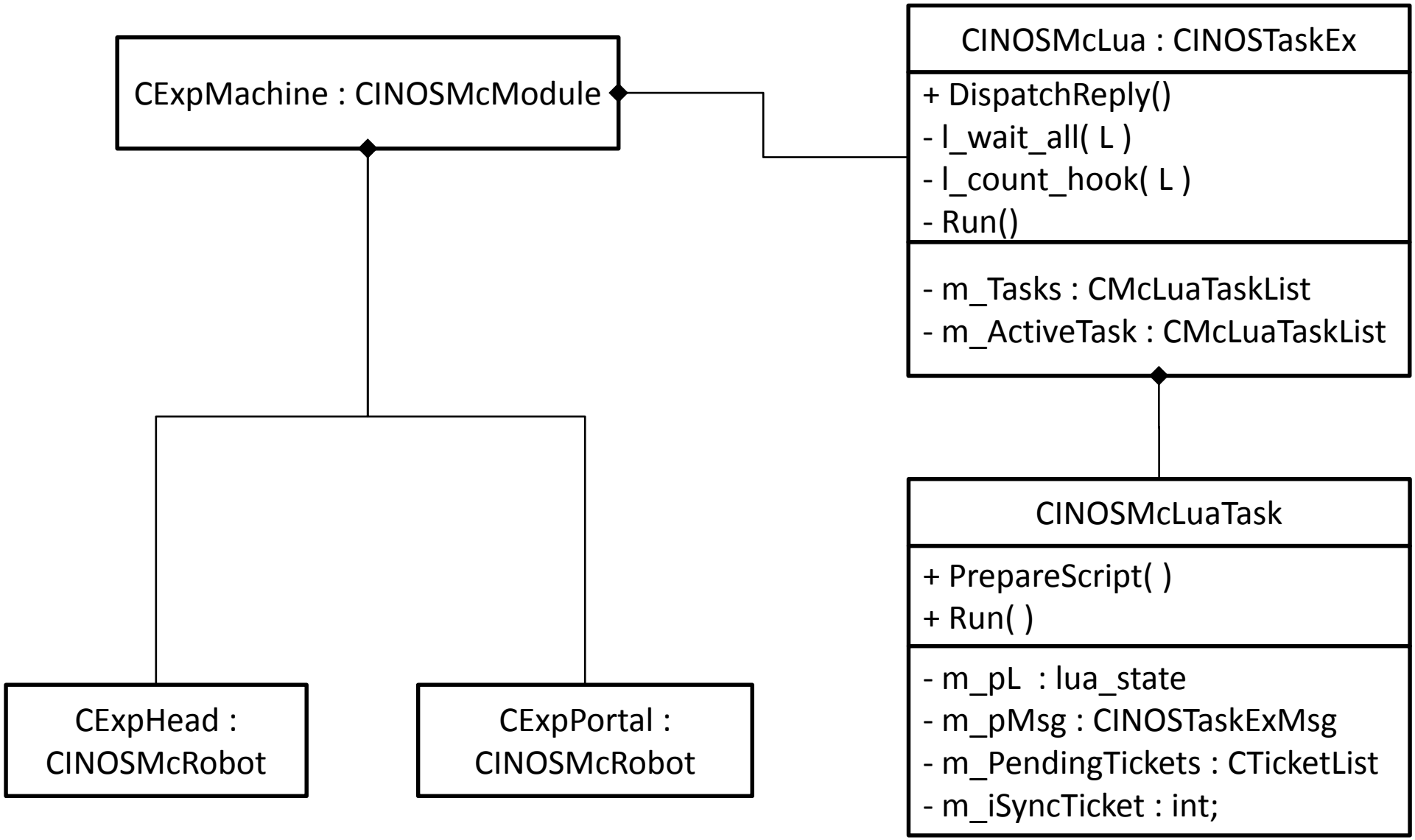
Why preemtion?

- One bad Lua script may not block other scripts:

```
while true do Something() end
```

- Fast response is crucial!

Several Lua tasks per C++ task



```
void CINOSMcLua::iRunScript (CINOSTaskExMsg* apMsg) {

        // Create new CINOSMcLuaTask
        CINOSMcLuaTask* pTask = new CINOSMcLuaTask(apMsg);

        // Prepate the new script
        if(pTask->PrepareScript(apMsg, m_pLuaMainState) {
                // Add the task to internal lists
                m_Tasks.Add(pTask);
                m_ActiveTasks.Add(pTask);

                // Run Lua interpreter it
                Run();
        }
}
```

```
Bool CINOSMcLuaTask::PrepareScript (CINOSTaskExMsg* apMsg,
                                    lua_state* apLuaMainState) {

        const char* pScript = apMsg->GetParam();

        // Create new Lua task
        lua_getglobal(apLuaMainState, "Tasks");
        m_pL = lua_newthread(apLuaMainState);
        lua_setfield(apLuaMainState, -2, m_cName);
        lua_pop(apLuaMainState, 1);

        // Load script
        if(luaL_loadstring(m_pL, apChunk)) {
                MsgError(apMsg, INOS_MCMSG_CODE_LUA_COMPILE_ERROR);
                return false;
        }
        // Defer in case script takes longer
        MsgDefer(apMsg);
        // Safe the message
        m_pMsg = apMsg;
        return true;
}
```

```
CINOSMcLua::Run() {

        // Cycle through all active Lua tasks
        m_pCurrentTask = m_ActiveTasks.First();
        while(m_pTask && GetMsgQueue->Empty()) {
                CINOSMcLuaTask* pNext = pTask->Next();
                // Run it
                m_pCurrentTask->Run();
                m_pCurrentTask = pNext;
        }
        m_pCurrentTask = NULL;

}
```

```
CINOSMcLuaTask::Run {
        // Run Lua
        switch(lua_resume(m_pL, 0, 0)) {
                case LUA_OK:
                        m_pLua->ActiveTasks.Remove(this);
                        m_pLua->Tasks.Remove(this);
                        MsgDone(apMsg);
                        return;
                case LUA_YIELD:
                        return;
                case LUA_ERRRUN:
                case LUA_ERRERR:
                        m_pLua->ActiveTasks.Remove(this);
                        m_pLua->Tasks.Remove(this);
                        MsgError(apMsg, INOS_MCMSG_CODE_LUA_RUNTIME_ERROR);
                        return;
                default: //shouldn't happen ☹
                        ASSERT_ALWAYS(false);

    }
}
```

```
// __call of "McModuleCmd" metatable
int l_call_command(lua_State *L) {
        (...)
        CINOSTaskExMsg* pCallMsg =
                new CINOSTaskExMsg( CINOSTaskEx::eMsgCmd,
                                    pCmdMsg->GetMsgCode(),
                                    DF_INOS_ASYNCHRONOUS); // Always async
        pCallMsg->SetUser(pMcLua->m_pCurrentTask);
        // Loop through the parameters and add them
        (...)
        // Put the message safe the ticket
        int ticket = pCmdMsg->GetTask()->PutMsg(pCallMsg,
                                                pCallMsg->GetId());
        pMcLua->m_PendingTickets->Add(ticket);

        // if synchronous, store this ticket to sync
        if(lua_tointeger(L, -1)  == CINOSMcLua::eCmdSync) {
                pMcLua->m_pCurrentTask->m_iSyncTicket = ticket;
                // make this task inactive and yield
                pMcLua->ActiveTasks.Remove(pMcLua->m_pCurrentTask);
                return lua_yield(L, 0);
        }
}
```

```
// Global Lua function WaitAll()
int l_wait_all(lua_State *L) {

        pMcLua->m_pCurrentTask->m_iSyncTicket = 0;
        // 0 is not a valid ticket

        pMcLua->ActiveTasks.Remove(pMcLua->m_pCurrentTask);
        return lua_yield(m_pL, 0);
}
```

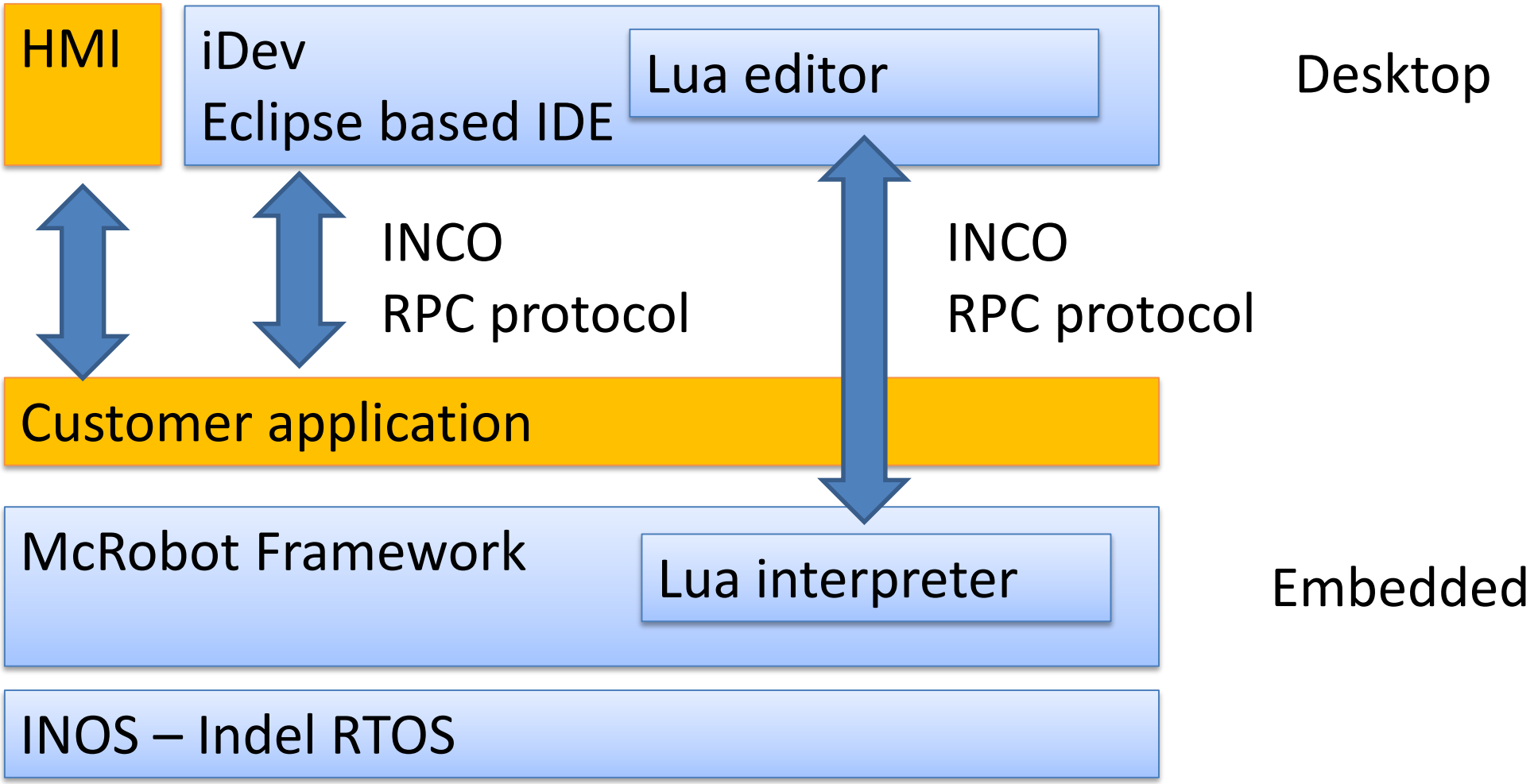
```
// Always called when a McRobot command has finished
CINOSMcLua::DispatchReply(CINOSTaskExMsg* apReply) {
    CINOSMcLuaTask* pTask = (CINOSMcLuaTask *) apRpl->GetUser();
    if(pTask != NULL) {
        // remove this ticked from the pending list
        pTask->m_PendingTickets.Remove(apReply->GetId());

        // restart task?
        if(apReply->GetId() == pTask-> m_iSyncTicket ||
           m_PendingTickets.Empty()) {

                // add task to scheduler
                m_pMcLua->m_ActiveTasks.Add(this);
        }
    }
    // run again!
    Run();
}
```

```
// count hook of Lua interpreter
int l_count_hook( lua_State *L ) {
	CINOSMcLua* pMcLua = (CINOSMcLua*) lua_getud(L);

	// Message pending?
	if(!pMcLua->GetMsgQueue()->Empty()) {
		return lua_yield(L, 0);
	}

	// Task ran for too long?
	if(pMcLua->ElapsedTime() >
				pMcLua->m_pCurLuaTask->m_TimeSlice) {
		return lua_yield(L, 0);
	}
}
```

The result

```
function Production()

        Mc.Module.Portal.Cmd.MoveXY(100.0, 120.0, Async)
        Mc.Module.Head.Cmd.MoveT(30.0, Async)
        WaitAll()                                          -- yields
        Mc.Module.Head.Cmd.Pick(12.0, Sync)                -- yields

        Mc.Module.Portal.Cmd.MoveXY(100.0, 120.0, Async)
        Mc.Module.Head.Cmd.MoveT(30.0, Async)
        WaitAll()                                          -- yields
        Mc.Module.Head.Cmd.Pick(12.0, Sync)                -- yields

end
```

Eclipse based Lua debugger


HMI
iDev
Eclipse based IDE
Lua editor
Desktop
INCO
RPC protocol
INCO
RPC protocol
Customer application
McRobot Framework
Lua interpreter
Embedded
INOS – Indel RTOS

What do we need?

- Lua editor resource for Eclipse
- Eclipse debug interface to INCO mapping
- INCO to Lua interpreter mapping
- No runtime impact for breakpoint support
- Minimal impact while debugging

→ Debugger must run in C/C++

Lua editor resource

Koneki

- Powerfull Lua IDE ☺
- But debugger does not do what I want to ☹
- I was unable to replace it ☹ ☹ ☹

EMFText generated Eclipse resource

- Generated parser for syntax highlighting
- Generated stubs for debug interface

Eclipse to INCO binding

Map each Eclipse debug function to one or several INCO calls

```
public void stopTask() throws org.eclipse.debug.core.DebugException {
        Inco_32.CallProcedure(luaIncoTarget, "Lua.Cmd.StopTask", name);
}
```

INCO to Lua interpreter mapping

Simply add a new McRobot command for each debugger function

```
CINOSMcLua::iStopTask(CINOSTaskExMsg* apMsg) {
	const char* pTaskName = apMsg->GetParam();
	m_ActiveTasks.Remove(pTaskName);
	pMcLua->m_pCurrentTask->m_eStatus == eHalt;
	MsgDone(apMsg);
}
```

No runtime impact breakpoint support

- New breakpoint opcode
- New hook for breakpoint hit
- Find first opcode for source line
- Replace opcode, save original opcode

```
// breakpoint hook
int l_breakpoint_hook(lua_State *L, lua_Debug *ar) {
	CINOSMcLua* pMcLua = (CINOSMcLua*) lua_getud(L);

	// suspend this task and yield!
	pMcLua->ActiveTasks.Remove(pMcLua->m_pCurrentTask);
	pMcLua->m_pCurrentTask->m_eStatus == eHalt;
	return lua_yield(L, 0);
}
```

Other Lua tasks still run when one tasks is on halt!

Cross-compiler library for Lua source to embedded bytecode

Linux

x86 AMD64
Little endian

Windows

PowerPC 750GX/GL
Big endian, with FPU

PowerPC 405 GPR
Big endian, without FPU

ARM Cotrex A9
Little endian, with FPU

ldump.c: Lua bytecode dumper

```
int luaU_dump (lua_State* L, const Proto* f, lua_Writer w, void* data,
               int strip)
{
     DumpState D;
     D.L=L;
     D.writer=w;
     D.data=data;
     D.strip=strip;
     // ...
}
```

Modified ldump.c

```
int luaU_dump (lua_State* L, const Proto* f, lua_Writer w, void* data,
               int strip, int fixedpoint, int switch_endianness)
{
     DumpState D;
     D.L=L;
     D.writer=w;
     D.data=data;
     D.strip=strip;
     D.fixedpoint=fixedpoint;
     D.switch_endianness=switch_endianness;
     // ...
}
```

Original ldump.c

```
#define DumpMem(b,n,size,D)        DumpBlock(b,(n)*(size),D)
```

Modified ldump.c

```
static void DumpMem (const void* b, int n, size_t size, DumpState* D)
{
  if(D->switch_endianness == 1) {
        char* cBuffer = malloc(n * size);

        SwitchEndianess(cBuffer, b, n, size);

        DumpBlock(cBuffer, number * size,D);
        free(cBuffer);
  }
  else {
        DumpBlock(b, n * size,D);
  }
}
```

Original ldump.c

```
static void DumpNumber(lua_Number x, DumpState* D)
{
 DumpVar(x,D);
}
```

Modified ldump.c

```
static void DumpNumber(lua_Number x, DumpState* D)
{
 if(D->fixedpoint) {
       int64_t fixed = (int64_t)(x * 4294967296.0);
       DumpVar(fixed,D);
 }
 else {
       DumpVar(x,D);
 }
}
```

What about different hosts?

- 32 / 64 bit systems have different size of size_t
- Linux and Windows have different definitions of 64 bit integer types

→ Use C99 type definition (int32_t, int64_t …)

Muito obrigado!