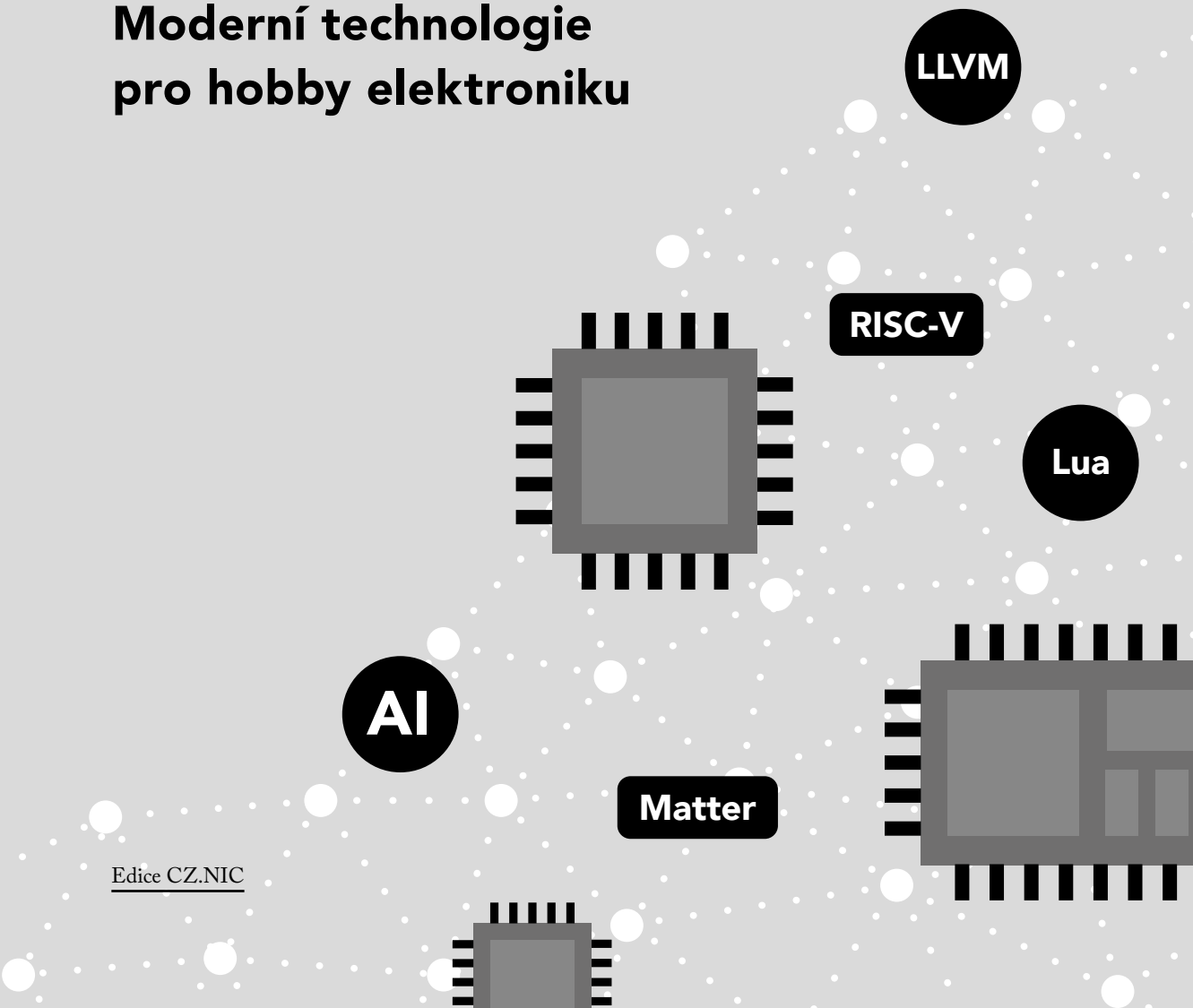
Martin Malý

Kity, bity, neurony

Moderní technologie pro hobby elektroniku



Edice CZ.NIC

KITY, BITY, NEURONY

Martin Malý

Vydavatel:
CZ.NIC, z. s. p. o.
Milešovská 5, 130 00 Praha 3
Edice CZ.NIC
www.nic.cz

1. vydání, Praha 2025
Kniha vyšla jako 35. publikace v Edici CZ.NIC.
ISBN 978-80-88168-85-0

ISBN 978-80-88168-85-0

— Martin Malý

Kity, Bity, Neurony

— Edice CZ.NIC

Předmluva vydavatele

Předmluva vydavatele

Milá čtenářko, milý čtenáři,

Vámi otvíraná kniha, nazvaná Kity, bity, neurony, představuje již šestou publikaci Martina Malého vydávanou v Edici CZ.NIC. Autor se tentokrát zamýšlí nad dynamickým vývojem v celém „oboru bastlení" – domácího kutilství. Ačkoliv se podle autora někteří čtenáři dříve dožadovali návratu k tranzistorům, kniha se místo nostalgie soustředí na to, co se ve světě hobby elektroniky objevilo relativně nedávno.

Cílem autora bylo ukázat nové možnosti domácího kutilství, představit zajímavé oblasti a být užitečným průvodcem v domácím bastlení zvláštnímu druhu člověka kutilu elektrotechnickému. Martin Malý se v knize postupně věnuje v jednotlivých kapitolách softwaru a hardwaru, modulům, praktickým aplikacím a v neposlední řadě umělé inteligenci a neuronovým sítím. Jako bonus si čtenář může postavit třeba bezdotykovou lampičku ovládanou gesty.

Kniha vzdává hold domácímu kutilství, které autor rehabilituje jako umění vytvořit něco vlastníma rukama, s maximem kreativity a radosti z objevování. Bastlení je totiž nutné vnímat ne jako způsob jak získat hotové bezchybné řešení, ale jako kreativní proces učení a radosti z řešení problémů. A právě radost z tvoření a touha porozumět všudypřítomným technologiím představují základní vlastnosti kutilského člověčenství. Přejeme vám příjemné čtení a neutuchající touhu stavět, objevovat a žít.

Jaromír Novák, CZ.NIC
Praha, 25. listopadu 2025

Poděkování

Poděkování

Když jsem psal poděkování do své první knihy, netušil jsem, že jednou budu psát poděkování do knihy šesté. Ale stalo se to, a to především díky mnoha skvělým lidem.

V první řadě děkuji lidem, kteří tuto knihu v Edici CZ.NIC připravili. Jmenovitě pak paní Kateřině Slavíkové, bez níž by tyto knihy ani nevznikaly. Na počátku je vždycky její telefonát s nenápadnou otázkou „…a nechcete zase něco napsat?“ Nedokážu odmítnout!

Děkuji lidem, kteří bastlení považují za radost, ne za výsadu, a kteří mi pomohli radou, tipem, námětem, nebo jen svým pozitivním příkladem. Pro mě jsou duší českého bastlení. Za všechny jmenuji Michala Ševčíka, Petra Šrámka, Martina Kiklhorna, Juraje Michálka, Konstantina Lásku, Oldřicha Horáčka, Martina Wolkera, Martina Kocourka, ale kdybych chtěl být spravedlivý, tak bych musel jmenovat všechny ty lidi, co znám z Twitteru (nyní X) jen přezdívkami a podle jejich profilových fotografií.

Úplně nejvíc pak děkuji vám všem, kteří moje knihy čtete a kupujete. Vždycky mě potěší, když se ke mně někdo hlásí a říká: „Jé, já četl vaši knížku!“ – a co teprve když je z oboru, kde bych to moc nečekal. Díky za tahle milá potěšení!

Speciální poděkování za toleranci a trpělivost patří mojí partnerce Lence.

Díky!

Předmluva

Předmluva

Napsal jsem pět knih o elektronice. Když jsem dopsal první, věděl jsem, že bude mít pokračování o mikroprocesorech a stavbě vlastních počítačů. Když jsem psal druhou, věděl jsem, že bude muset být třetí, o FPGA. A pak jsem myslel, že už jsem napsal všechno, co jsem chtěl o elektronice napsat…

Jenže díky vám, čtenářům, vzniklo i speciální „školní" pojednání o Micro:bitu, a když jsem ho dopisoval, přišly nové modely ESP32, spousta konstrukcí, a bylo jasno: další kniha musí být o ESP32!

I po páté knize jsem si říkal, že už asi nebudu mít o čem psát, ale obor jde rychle dopředu. Nemyslím elektroniku, tam je to nepochybné; myslím „obor bastlení", domácího elektronického kutilství. Když se mi opět ozval vydavatel s tím, jestli nechci napsat další knihu, tak jsem se zamyslel: o čem by měla být?

Někteří navrhovali vrátit se k tranzistorům. Jiní navrhovali rádio. Obojí jsou zajímavé oblasti. Ano, uznávám, že „bastlení" s tranzistory má svoje kouzlo, i když leckdy nostalgické: vůně pálené kalafuny, olověná pachuť na jazyku, tranzistory s dlouhými nožičkami rozkročené nad změtí drátků, v lepším případě nad deskou plošných spojů, měření, nastavování pracovního bodu zesilovače, vinutí cívek („viňte 624 závitů vodičem 0,1 mm a každou vrstvu proložte papírem. U závitu 218 udělejte odbočku. Na cívku naviňte 12,5 závitu lakovaným vodičem 1 mm" – pamatujete?) a oživování toho všeho… Tranzistory mi voní zažloutlými listy knih, které jsem četl a miloval jako kluk, ale svět se za těch skoro padesát let posunul. Když dnes postavíte rádio tak, jak se stavělo „tenkrát", moc toho k poslouchání nechytíte. Většina vysílání je už FM nebo DAB+ a to je už daleko za „tranzistorovým" obzorem. Samozřejmě si můžete postavit přijímač s FM i DAB, kde bude anténa, trocha bižuterie a jeden integrovaný obvod, co se o všechno stará, ale zas tam chybí ta tranzistorová romantika.

Nakonec jsem si říkal, že tranzistory jsou „nyní již nestárnoucí" a vrátit se k nim mohu kdykoli. A když jsem se rozhlížel, co se ve světě hobby elektroniky děje, tak mi došlo, že se musím trošku vrátit ke konceptu první trilogie: základní principy, ukázky zapojení, různé možnosti, novinky, techniky.

Tato kniha je věnovaná věcem, které se objevily v bastlířském světě relativně nedávno. Rozdělil jsem ji do čtyř hlavních částí: software, hardware, moduly a AI, respektive neuronové sítě.

V oblasti software se objevují nové jazyky, které v některých situacích nabourávají téměř absolutní dominanci C/C++. Kromě Pythonu a jeho variant pro MCU jde třeba o skriptovací jazyk Lua, který se hodí pro psaní různých pluginů a uživatelských kódů, nebo o „bare metal" verzi Rustu, což je jazyk s velmi pomalou učící křivkou, ale na konci jste odměněni velmi bezpečnými programy, protože vás už samotný překladač chrání od konstrukcí, ve kterých by mohlo docházet k proble-

mům. Zařadil jsem i nový jazyk Zig, který je zajímavou alternativou k C a zároveň s ním dokáže skvěle koexistovat. Popíšeme si i principy LLVM, který se stal základem mnoha překladačů.

Hardwarová část obsahuje seznámení s Raspberry Pi Pico 2, s velmi levným MCU CH32V003, se zajímavým duálním Milk-V Duo, a přidal jsem i novou generaci ESP32. Když se na tyto obvody podíváte s odstupem, uvidíte, že je jedna věc spojuje, a to je architektura RISC-V. Tato modulární architektura, odvozená ze známé architektury MIPS, je open-source, takže ji výrobci čipů mohou používat ve svých zařízeních bez nutnosti platit licenční poplatky. To spolu s bohatými možnostmi přizpůsobení z ní dělá zajímavou a perspektivní alternativu například k ARM. Proto jsem velkou část věnoval popisu této architektury.

Ve třetí části se budu věnovat praktickým věcem, u kterých jsem si říkal, že by měly zaznít. Například jak ve vlastních konstrukcích vyřešit napájení přes stále populárnější USB-C, jaké zvolit čidlo a jaké jsou možnosti, a popisu některých projektů, které jsou zajímavé i jako inspirace k vlastní tvorbě.

A poslední část knihy věnuji neuronovým sítím. Je bez debat, že tato technologie bude v dalších letech téměř všude, a díky rozvoji elektroniky bude stále častěji dostupná i pro domácí hobby projekty. Mnoho čipů už dnes nabízí různé „AI akcelerátory“ a koprocesory, proto jsem zařadil tuto sekci. Popíšeme si základní principy neuronových sítí (perceptron, vícevrstvé sítě, backpropagation, učení), jejich použití v mikrokontrolérových zařízeních (kvantizace sítě pro malé procesory), používané knihovny a příklad toho, jak spustit na malém jednočipu vlastní neuronovou síť.

Dlouhé zdrojové kódy k příkladům a užitečné odkazy najdete na https://kbn.elektroniche.cz

Obsah

1 Chvála bastlení

1 Chvála bastlení

V této knize vzdávám hold domácímu kutilství, zaměřenému na elektroniku. Rukopis dokončuji na podzim roku 2025, a říkám si, že je to docela dobrá příležitost podívat se na „posledních sto let s elektronikou“ očima domácích kutilů. I proto jednotlivé části knihy oddělují historická intermezza, popisující jednotlivá čtvrtstoletí, nejprve s elektronkami, pak s tranzistory, poté s integrovanými obvody, dnes s jednočipovými mikrokontroléry.

Pro podobné činnosti se vžilo označení „bastlení“, které je trochu pejorativní, ale já bych ho rád alespoň pro tuto knihu rehabilitoval.

„To je ale zbastlené,“ říkávali starší a kroutili hlavou nad poličkou, která se mírně nakláněla doleva. V jejich hlase bylo znát zklamání a slovo „zbastlit“ znělo z jejich úst jako obžaloba. Jako když řekneme, že je něco odfláknuté, nedodělané, provizorní. V běžné řeči se bastlení stalo synonymem pro provizorní řešení, hala bala, nakřivo, polorozpadlé, podepřené kusem kartonu a zafixované izolepou.

Jenže pak je tu ještě jiné bastlení. To, které znamená pravý opak. Bastlení jako umění vytvořit něco vlastníma rukama, s minimem prostředků, ale s maximem kreativity. Bastlení jako proces objevování, učení se a radosti z tvoření.

Každý bastlíř to zná, ten moment, kdy se v garáži nebo u pracovního stolu rodí něco nového. Není to proto, že by to nutně potřeboval. Není to proto, že by si to nemohl koupit hotové. Je to proto, že chce pochopit, jak věci fungují. Chce jim dát něco ze sebe. A především si chce užít tvorbu.

Bastlení je jako jazz. Můžete poslouchat dokonale nahranou studiovou skladbu, nebo se můžete posadit k pianinu a začít experimentovat. Bude to zpočátku skřípat, některé tóny nebudou sedět, ale bude to vaše. A časem, s každým dalším pokusem, s každou další hodinou strávenou učením a zkoušením, se začne rodit něco jedinečného.

V době, kdy máme všechno na dosah ruky, kdy stačí několik kliknutí k objednání prakticky čehokoliv, může bastlení působit jako přežitek. Proč trávit hodiny stavbou vlastního teploměru, když si můžeme koupit profesionální meteostanici? Proč pájet vlastní zesilovač, když jsou k dispozici špičkové audio systémy? Proč programovat vlastní domácí automatizaci, když existují hotová řešení?

Odpověď je jednoduchá: Protože bastlení není způsob, jak mít systém pro automatizaci, meteostanici nebo osvětlení. Podstatný je proces, ta zvláštní magie, která se děje, když vezmete do ruky pájku, když poprvé připojíte nový senzor, když váš vlastnoručně napsaný program konečně udělá přesně to, co jste chtěli. Nebo když něco nefunguje a vy musíte přijít na to proč. Bastlení je radost z řešení problémů a uspokojení z překonávání překážek.

Bastlení je také komunitní věc. Spojuje lidi, kteří si navzájem pomáhají, sdílejí své zkušenosti, radí se a inspirují se. Bastlení jsou i večery strávené na internetu hledáním řešení zdánlivě neřešitelného problému, ale i nadšení, když konečně najdete někoho, kdo řešil něco podobného. Je to sdílená radost z úspěchu i společném učení se z neúspěchů.

V době, kdy se věci stávají stále více uzavřenými a nepřístupnými běžnému člověku, kdy jsou naše zařízení černými skříňkami, do kterých nemůžeme nahlédnout, je bastlení aktem svobody. Je to způsob, jak si zachovat kontrolu nad technologiemi, které nás obklopují. Jak jim porozumět, jak je přizpůsobit našim potřebám, jak je učinit opravdu našimi.

Takže až příště někdo řekne, že něco je „zbastlené", možná je čas se zamyslet. Možná je čas připomenout, že bastlení je ve skutečnosti projevem té nejvyšší péče a lásky k řemeslu. Je to způsob, jak věcem vdechnout duši, jak jim dát příběh. A především je to způsob, jak zůstat zvídavými dětmi v těle dospělých, jak si zachovat radost z objevování a tvoření.

Protože bastlení není jen sestavování elektroniky. Je to svoboda tvořit, odvaha experimentovat a radost z poznávání. Bastlení v nás otevírá to nejlepší, co v nás je: tvořivost, zvídavost a nekonečnou touhu učit se nové věci.

A možná právě proto si zaslouží, abychom ho psali s velkým B.

2 Software

2 Software

2.1 Mikrokontroléry nejsou jen C/C++ a Python

Embedded vývoji (programování vestavěných systémů) tradičně dominuje jazyk C/C++. Je z povahy nízkoúrovňový a má malou režii. V současnosti však existují i modernější alternativy zaměřené na vyšší spolehlivost, produktivitu či lepší správu paměti, které můžete použít na platformách jako Arduino (AVR), ESP32 (Xtensa/RISC-V), RP2040 (ARM Cortex-M0+) nebo RP2350. Představíme si dva takové jazyky, Nim a FORTH, z pohledu jejich současného využití v takových zařízeních.

V dalších kapitolách si podrobněji probereme jazyky Lua a Zig. Jeden z nich je vhodný pro aplikace, kde chceme dát možnost uživatelům psát vlastní bezpečné skripty a provádět je přímo na zařízení, druhý je zajímavou alternativou k C, která přidává nové vlastnosti, ale nerozbíjí kompatibilitu.

Taky se budeme věnovat **toolchainům**, tedy řetězcům nástrojů, které jsou spolu provázané a zpravidla jeden nástroj používá výstup druhého. Typicky jde o řetězec s navazujícími komponentami: překladač do nižšího jazyka → překladač do objektového kódu → spojovací program (linker). I když se volají třeba jedním příkazem, tak tam stále někde v pozadí je spousta nástrojů…

2.1.1 Nim

Nim (dříve Nimrod) je vyšší programovací jazyk, který přináší do systémového programování syntaktickou lehkost podobnou Pythonu, ale kompiluje se do efektivního nativního kódu (přes C/C++, nebo přímo). Oficiálně je prezentován jako *„staticky typovaný kompilovaný systémový programovací jazyk“*, který kombinuje osvědčené koncepty z Pythonu, Ady a Moduly-2. Je to jazyk multiparadigmatický: podporuje imperativní strukturovaný styl, funkcionální prvky (vyšší funkce, lambda výrazy), objektově orientované programování (dědičnost, metody) i metaprogramování pomocí maker.

Nim je navržen s ohledem na čitelnost a expresivitu kódu: syntaxi má podobnou Pythonu (výrazný rys je např. odsazovací bloková struktura místo složených závorek), ale zároveň nabízí nízkoúrovňové konstrukce (např. ukazatele, bitovou aritmetiku), potřebné pro systémové programování. Díky této kombinaci je Nim někdy označován za jazyk, který propojuje skriptovací jazyky a výkonné kompilované jazyky.

Hlavní rysy jazyka

Nim klade důraz na efektivitu kódu a flexibilní správu paměti. Výchozí implementace překládá Nim do jazyka C/C++ (případně do Javascriptu pro web), jedná se tedy o **transpiler**. Zkompilované programy nevyžadují žádný virtuální stroj ani runtime prostředí nebo knihovny. Výsledkem je nativní samostatný spustitelný soubor.

Nim nabízí několik režimů správy paměti. Tradičně používal garbage collector (nepřetržitý běhový GC, podobně jako Go nebo Java), ovšem novější verze zavedly deterministickou správu paměti pomocí destruktorů a přemisťovací sémantiky, inspirovanou C++ a Rustem. To znamená, že v moderním Nimu (od verze 1.4) můžete využít tzv. *ARC (automatic reference counting)* nebo *ORC*, tj. automatické počítání referencí na objekt s předvídatelným uvolňováním objektů ve chvíli, kdy už na ně neexistuje žádný odkaz. Pro MCU je to zásadní, protože Nim umí pracovat zcela bez garbage collectoru. Stačí přepnout režim (`--gc:arc` nebo `--gc:none` při kompilaci). V režimu `--gc:none` je odpovědnost za uvolňování paměti plně na programátorovi (podobně jako v C), nicméně díky tomu můžete využít Nim i v systémech s extrémně malou pamětí. Ale i s garbage collectorem se Nim snaží být šetrný. ARC dealokuje paměť průběžně a nemá pauzy, tzv. *stop-the-world*, jako jiné garbage collectory.

Silnou stránkou Nimu je jeho systém maker. V Nimu můžete psát makra, která pracují přímo se syntaktickým stromem (AST) programu. Tím můžete v jazyce definovat nové konstrukce, doménově specifická rozšíření apod., aniž by se měnila syntaxe jazyka (makra v Nim využívají stávající syntaxi, která je dostatečně flexibilní). To je výhodné třeba pro generování ovladačů z popisu periferie. Lze si nadefinovat makra, která na vstupu dostanou popis periferie a vygenerují potřebné konstanty a přístupové funkce.

Nim má moderní typový systém s inferenčními schopnostmi (mnohé proměnné nemusí mít explicitně uvedený typ, odvodí se), podporuje *tuples* (n-tice), *generika* (šablony) i *sum types* (sjednocené datové typy podobné variantám/algebraickým typům). Například zpracování chyb může Nim řešit jak výjimkami, tak pomocí sum typů (třeba návratová hodnota může být `Result[T, Error]` podobně jako v Rustu).

Nim je velmi dobře kompatibilní s jazyky **C/C++**. Protože kompiluje do jazyka C, tak je snadné volat z kódu v Nim libovolnou funkci v C nebo využít existující knihovnu. Typy můžete přebírat automaticky a volání je bez režie. Můžete přímo zahrnout hlavičkový soubor od výrobce (např. mapu registrů pro specifickou periferii) a přistupovat k němu z Nim.

V podstatě platí, že *jakákoli platforma, která má překladač C, je podporována Nimem*. Nim je v tomto skutečně univerzální. Může generovat kód pro AVR, ARM, ESP32, cokoliv, pokud k tomu existuje odpovídající překladač C.

Runtime a knihovny

Výsledné spustitelné soubory v Nimu jsou typicky malé a samostatné. Standardní knihovna je poměrně bohatá (obsahuje například moduly pro datové struktury, formátování textu, síťovou komunikaci atd.), ale v mikrokontroléru by se použila minimálně nebo vůbec. Základní runtime Nimu může obsahovat podporu pro garbage collector (pokud je povolený), případně několik pomocných rutin (např. pro aritmetiku nad velkými celými čísly, pokud jste ji použili). Lze ho však výrazně omezit. Vhodnou konfigurací překladače (pomocí konfiguračních souborů `.nims`) můžete vypnout generování výpisu zásobníku, odkazů pro ladění atd., a ušetřit další paměť. V extrémním případě (ARC/žádný GC, vypnutý debug) se velikost programu blíží holému kódu v C, potřebuje pouze základní věci jako funkce `memset`/`memcpy` (a i ty si může případně sám vygenerovat).

Existují projekty, napsané v Nimu pro osmibitový AVR s pouhými 2 kB RAM. Samotné jádro Nimu v nich zabírá jen jednotky kilobajtů. Pro 32bitové MCU (ARM, ESP32) nejsou zdroje tak kritické, takže je možné použít i komfortnější správu paměti (ARC). Ta přidá nějaký kód navíc (např. destruktory a počítadla referencí ke každému alokovanému objektu), ale stále jde o relativně malou režii v desítkách kB.

Žádné externí závislosti, kromě standardní knihovny céčka (pokud je využita pro volání `malloc` atd.) nejsou nutné.

Ekosystém

Nim má aktivní komunitu, ačkoli menší než populárnější jazyky. Distribuován je s vlastním balíčkovacím manažerem **Nimble**, kde můžete najít množství knihoven od webových frameworků po utility. Pro vývoj s MCU není zatím moc specializovaných knihoven, ale existují zajímavé projekty:

- **Nesper:** balíček pro oficiální ESP-IDF (Espressif IoT Development Framework), ve kterém můžete psát firmware pro ESP32 v Nim a využívat všech funkcí wifi, Bluetooth atd. (viz projekt `elcritch/nesper` na GitHubu). Nesper demonstruje silnou stránku Nimu, totiž plnou kompatibilitu s existujícími API v C. Kód v Nesperu volá funkce knihovny od Espressif bez jakéhokoli přidaného kódu, jen s trochou syntaktického cukru.
- **CMSIS/NXP v Nimu:** díky snadnému importu hlavičkových souborů z C můžete v Nim rychle získat definice registrů pro ARM MCU (CMSIS) a pracovat s nimi. Někteří vývojáři sdíleli `.nim` soubory vygenerované ze SVD popisů.
- Existuje např. i projekt využívající Nim v malé satelitní platformě (**CubeSat**), kde byl vybrán díky nabízené kombinaci výkonu a vysoké abstrakci při analýze dat.

Překladač Nimu je velmi rychlý a podporuje i interpretaci *NimScriptu*. To se využívá při konfiguraci. Co se týče IDE, existují pluginy pro VSCode, JetBrains IDE a další, s podporou syntaxe

a základního doplňování. Kód můžete ladit na úrovni vygenerovaného C, tedy ladit výsledný program pomocí gdb jako běžný program v C (je to málo komfortní, ale možné). Pro běžný vývoj pro mikrokontroléry se Nim integruje například do PlatformIO.

Ekosystém knihoven pro periferie není v Nimu centralizovaný. Vývojář typicky využije existující knihovnu v C. Nim se často označuje za *„hostovaný jazyk"* (hosted language), to znamená, že se při některých nízkoúrovňových operacích spoléhá na hostitelský jazyk, typicky céčko. To ale v kontextu malých zařízení nevadí, spíš to usnadňuje integraci se stávajícími vývojovými balíky pro jednotlivé platformy.

Celkově je ekosystém Nimu pro vývoj jednočipových aplikací zatím skromný, ale díky kompatibilitě s C není třeba vynalézat kolo. Kdokoli, kdo zvládne zkompilovat projekt v C pro daný mikrokontrolér, může totéž udělat s projektem v Nimu. Nim je tak pro MCU spíše „toolbox", který využívá existující nástroje. Jeho výhodou je, že není omezen architekturou.

Vhodné typy mikrokontrolérových projektů

Nim se hodí pro projekty, kde je požadována vyšší abstrakce a produktivita, než nabízí C, ale současně je potřeba zachovat relativně nízkou režii a přístup k hardware. Některé scénáře:

- **Hobby a IoT projekty na výkonnějších MCU:** Například ESP32 má desítky až stovky kB RAM, a to je dostatečné pro běh Nimu s ARC. Nim umožní rychle vyvinout aplikaci (např. čidlo s web serverem) s mnohem menším množstvím nutného kódu než při psaní v čistém C/C++. Jako programátor můžete využít vyšší datové struktury, generické funkce, serializaci JSON apod. bez velkých starostí o paměť, protože ARC se postará o její správné uvolnění.
- **Prototypování algoritmů:** Nim svou syntaxí podobnou Pythonu láká k rychlému psaní algoritmů, které pak běží nativně. Pokud vyvíjíte třeba zpracování signálu nebo řídicí algoritmus pro ARM Cortex-M4, v Nimu ho napíšete téměř jako pseudokód, ale výsledek poběží s výkonem C. Makra umožní vkládat ladicí výstupy, generovat opakující se části atd.
- **Vestavěné aplikace s komplexní logikou:** Stavíte např. datalogger, který komunikuje přes různé protokoly, ukládá data na SD kartu, reaguje na události. V Nimu můžete takový firmware strukturovat modulárně (použít objekty pro abstrakce zařízení), využít výjimky pro chybové stavy atd. Zjednoduší to kód oproti C, kde by se muselo ručně kontrolovat každé volání funkce (v Nimu lze použít try/except podobně jako v Pythonu, ale s minimální režií).
- **Multiplatformní logika:** Pokud část kódu má běžet i na PC i na MCU (např. nějaká knihovna sdílená mezi firmware a simulačním modelem na PC), tak Nim je výborná volba. Stejný zdrojový kód můžete zkompilovat do JavaScriptu (pro webovou simulaci), do nativní aplikace i do firmware (přes C). Nemusíte psát výrazně odlišný kód pro různá prostředí.

Omezení tu samozřejmě jsou. Nim není ideální pro úplně nejmenší osmibitové mikrokontroléry s pár kB paměti, pokud bychom chtěli využívat komfortní funkce (dynamické řetězce, haldy). Sice

se dá i tam nasadit Nim (s přepínačem `--gc:none`), ale prakticky byste museli psát kód sice v Nimu, ale „jako v C“, takže by výhoda jazyka zčásti padla. Pro 32bitové MCU s desítkami kB paměti už ale Nim pohodlně funguje.

Další omezení je menší komunita. Na problém možná nenajdete odpověď tak rychle jako u C či Arduina. Také nástroje (ladění, trasování) nejsou tak vyspělé. Neexistuje ekvivalent JTAG debuggeru, který by byl integrovaný přímo s jazykem Nim. Používají se standardní nástroje pro C, ale to může být někdy nepohodlné kvůli odlišnostem.

Ukázka „Hello World“

Připomeňme si, že **Nim můžete spustit i na Arduino UNO (ATmega328P),** které má pouhých 2 kB RAM. Kompilátor to zvládne díky možnosti vypnout runtime a přizpůsobit se dané architektuře. Konfigurační soubor pro překlad může vypadat např. takto:

```
# config.nims – nastavení pro AVR
switch("os", "standalone")    # žádný OS
switch("cpu", "avr")          # cílová CPU architektura
switch("gc", "none")          # žádný garbage collector
switch("stackTrace", "off")   # nevkládat stack trace info
switch("lineTrace", "off")    # nevkládat debug info o řádcích
switch("passC", "-mmcu=atmega328p")   # parametry pro avr-gcc (překlad pro MCU)
switch("passL", "-mmcu=atmega328p")   # parametry pro linker
```

Takto zajistíme, že Nim vygeneruje C kód pro AVR a použije pro kompilaci AVR-GCC. Samotný kód v Nimu pak může přímo manipulovat registry MCU. Např. rozblikání LED na pinu 13 (Arduino UNO) by šlo takto:

```
const DDRB* = cast[ptr uint8](0x24)  # Data Direction Register B (ATmega328P)
const PORTB* = cast[ptr uint8](0x25) # Port B Data Register
proc delay_ms*(ms: int) =
  ## jednoduchá prodleva pomocí smyčky
  for i in 0 ..< ms*1000: discard       # (na 16MHz CPU ~ orientační čekání)

# Inicializace – nastav PB5 (arduino pin13) jako výstup
DDRB[] = DDRB[] or 0b0010_0000
while true:
  PORTB[] = PORTB[] xor 0b0010_0000    # přepni bit LED
  delay_ms(500)
```

Výše uvedený kód je ilustrační. Ukazuje přístup k paměťovým adresám (pomocí `cast[ptr uint8]`) a nekonečnou smyčku blikání. Přeložený program můžete nahrát do Arduina standardním postupem (např. pomocí `avr-objcopy` převést na .hex a nahrát pomocí `avrdude`). Je tu hezky vidět, že Nim je syntakticky mnohem stručnější než ekvivalent v C (není třeba psát datové typy u literálů, cyklus lze udělat pythonovským zápisem atd.), a přitom výsledek po překladu je velmi podobný strojovému kódu.

Dostupná literatura a online zdroje

- **Oficiální příručka Nim:** dokumentace na nim-lang.org obsahuje manuál popisující jazyk a standardní knihovnu. Pro vývoj elektroniky jsou důležité hlavně kapitoly o správě paměti a integraci s C.
- **Kniha „Nim in Action"** (Dominik Picheta, Manning 2017): úvod do jazyka a praktické projekty. I když není zaměřena přímo na MCU, tak pokrývá podrobně volání kódu v C a výkonové aspekty.
- **Nim community forum** (forum.nim-lang.org): aktivní fórum, kde najdete vlákna o použití Nim na ESP32, Arduino a dalších. Často tam přispívají samotní tvůrci jazyka.
- **Článek „nim for embedded software development"** (na Dev.to): zkušenosti vývojáře s použitím Nim pro MCU, včetně ukázky nastavení pro AVR.
- **Repozitář Awesome Nim:** komunitní seznam knihoven a projektů (GitHub `nim-lang/awesome-nim`), který zmiňuje i využití Nimu pro elektroniku.

2.1.2 Forth

Forth je nestárnoucí klasika mezi programovacími jazyky pro systémy s malými procesory. Jde o *zásobníkově orientovaný* jazyk a zároveň jednoduché interaktivní vývojové prostředí, který navrhl v 60. letech Chuck Moore a který byl hojně využíván v 70. až 90. letech v průmyslových vestavěných aplikacích. Paradigma Forthu je ojedinělé: je to konkatenativní (skládající) jazyk, kde se program skládá z posloupnosti slov (tokenů), které operují nad společným datovým zásobníkem. Forth nezná klasické proměnné a datové typy v běžném smyslu (vše jsou jen čísla na zásobníku), nemá syntaktická omezení (řídicí struktury používají speciální slova). Můžeme ho označit za procedurální jazyk bez kontroly typů (přesto se dá ve Forthu definovat např. oddělený zásobník pro desetinná čísla atd., ale to je implementační detail).

Důležitou vlastností je, že Forth typicky běží s interaktivním interpretem, je tedy zároveň vývojovým prostředím (IDE), kde uživatel může za běhu systému zadávat příkazy, zkoušet funkce atd.

Forth je výrazně odlišný od mainstreamových jazyků. Je postaven na RPN (obrácená polská notace, reverse polish notation), kde se zapisují nejprve operandy, a za ně operace. Všechny operace

berou parametry ze zásobníku a výsledek ukládají zpět. Např. součet 2 a 3 se zapíše jako `2 3 +` (kde „plus“ je slovo, které vezme dvě čísla ze zásobníku, sečte a výsledek uloží opět na zásobník).

Ve Forthu neexistuje syntaktický rozdíl mezi „jazykem“ a „vestavěnými funkcemi“, protože celé prostředí Forthu je definováno souborem slov (včetně řídicích struktur jako `IF..ELSE..THEN`, smyček `DO..LOOP` atd.). Uživatel může za běhu definovat nová slova (vlastní funkce), a ta se okamžitě stávají součástí jazyka. Forth je tedy extrémně flexibilní a rozšiřitelný; zkušený programátor si může přizpůsobit celé prostředí podle potřeby. Z jiných jazyků má blízko snad jen k PostScriptu (který je také zásobníkový) nebo modernějším jazykům jako Factor, ale v mnoha ohledech je Forth skutečně unikátní.

Hlavní rysy jazyka

- **Interaktivnost:** Forth typicky běží na cílovém zařízení jako interpret, ke kterému je možné se připojit (např. přes sériovou linku) a zadávat příkazy. Okamžitě po zadání příkazu je vidět výsledek. To urychluje vývoj a ladění; nepotřebujete plný debugger, stačí konzole. Proto se Forth používá často i jako zabudovaný monitor v průmyslových zařízeních.
- **Kompilace a interpretace:** Forth používá tzv. *threaded code*, kde se nové slovo (funkce) kompiluje jako sekvence volání jiných slov. Kompilace je velmi rychlá a probíhá přímo na zařízení (tzv. self-hosted). Z pohledu uživatele se střídají dva režimy: *interpretation state* (příkazy se ihned vykonávají) a *compile state* (příkazy se místo vykonání kompilují do definice nového slova). Přepnutí do *compile state* nastane po zadání slova `:` (dvojtečka, tedy začátek definice slova) a zpět do interpretace se přejde po `;` (středník, konec definice). Tato dualita dává Forthu schopnost definovat slova za běhu a hned je volat.
- **Extrémní efektivita a nízké nároky:** Implementace Forthu jsou velmi malé. Jednoduché jádro Forthu (interpreter + kompilátor + základní slovník) se vejde do několika kilobajtů paměti. Například **AmForth** pro mikrokontroléry AVR8 potřebuje jen ~8–12 KB flash a ~200 bajtů RAM pro běh základního systému. Na druhou stranu Forth nemá komfortní automatickou správu paměti, vše musíte udělat ručně (ale v malých MCU to nevadí, tam paměťovou *haldu* často nepotřebujete vůbec).
- **Jednoduchost běhového prostředí:** Klasický Forth nepotřebuje žádný operační systém ani volání služeb. Může běžet přímo od resetu MCU jako holý firmware. Implementuje si často i vlastní miniaturní operační systém (např. jednoduchý plánovač úloh, souborový systém v paměti atd.) Mnohé implementace Forthu také podporují multitasking (kooperativní nebo preemptivní) uvnitř sebe sama, i bez podpory ze strany OS. To znamená, že jediné, co musíte na MCU zajistit, je nastavení ukazatele zásobníku a skok na start Forth VM.
- **Přenositelnost a standardizace:** Forth byl formalizován v ANSI standardu (ANS Forth 1994, později Forth 200x). Díky tomu existuje množství implementací, které jsou do velké míry kompatibilní. Programy napsané v mezích standardu půjde zprovoznit na různých systémech s minimem úprav. Pro MCU se ale často používají *podstandardní implementace* (např. minimální jádro bez některých složitých slov kvůli úspoře paměti).