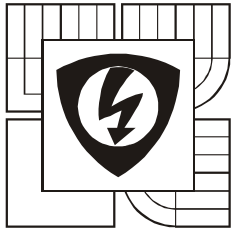
Vyšší programovací jazyky



Kurz: Signálové procesory

Autor: Petr Sysel

Lektor: Petr Sysel

Vytvoření této videopřednášky bylo podpořeno projektem č. CZ.1.07/2.2.00/28.0098
Evropského sociálního fondu a státním rozpočtem České republiky.

Obsah přednášky

Zápis programu a průběh překladu
Preprocesor
- Direktivy preprocesoru
- Předdefinované symboly
- Makra preprocesoru

Rozdělení programu do modulů
Datové typy
Použití instrukcí asembleru
- Intrinsic funkce

Implementačně závislé příkazy
Sestavovací program

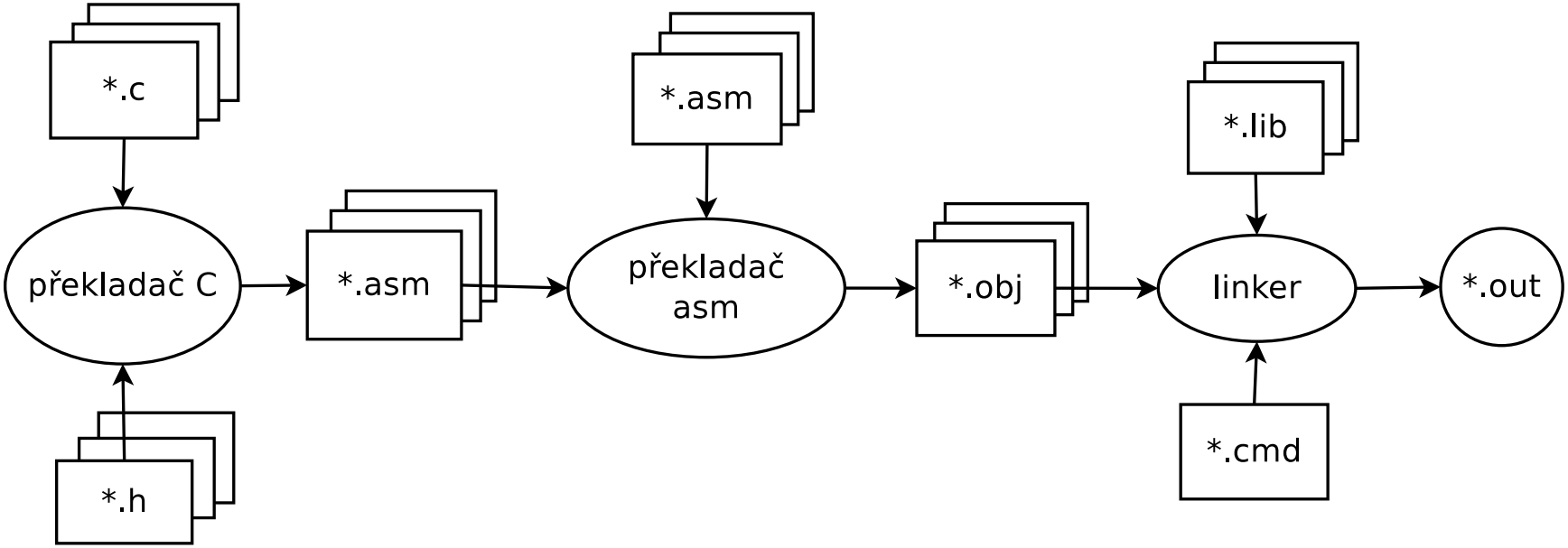
Motto:

Programování je proces
postupného nahrazování
předchozích chyb a omylů
mnohem důmyslnějšími
chybami a omyly.

Vyšší programovací jazyky

- Pouze prostředek pro vyjádření myšlenek,
- u embedded systémů se nejčastěji používá jazyk C, který je nízkoúrovňový ale výpočetně nenáročný,
- navržen v Bellových laboratořích AT&T 1969–1973 pro potřeby operačního systému Unix,
- většina překladačů pro signálové procesory podporuje ANSI C standard z roku 1989 (podpora C++ se může lišit),
- pro potřeby signálových procesorů je však nutné řešit některé problémy:
  - přizpůsobení datových typů,
  - volání speciálních instrukcí v asembleru,
  - definice dodatečných informací pro optimalizaci překladu.

Průběh překladu



Na obrázku chybí ještě preprocesor jazyka C nebo asembleru.

Direktivy preprocesoru

- Direktivy preprocesoru jsou většinou uvozeny znakem #,
- vkládání zdrojových (hlavičkových) souborů – `#include`,
- definice symbolů preprocesoru – `#define`,
- podmíněný překlad
  `#if` – `#else` – `#endif`,
  `#ifdef` – `#else` – `#endif`
  `#ifndef` – `#else` – `#endif`
  `#ifndef` – `#elif` – `#else` – `#endif`
- ukončení překladu – `#error`.

Předdefinované symboly

- `__DATE__` – datum překladu ve formátu `mmm dd yyyy`,
- `__TIME__` – čas překladu ve formátu `hh:mm:ss`,
- `__FILE__` – název aktuálně překládaného souboru,
- `__LINE__` – číslo aktuálního řádku,
- `__LITTLE_ENDIAN__` – definováno, pokud je překladač nastaven na LITTLE ENDIAN,
- `__BIG_ENDIAN__` – definováno, pokud je překladač nastaven na BIG ENDIAN,
- `_TMS320C6X` – definováno, pokud je použit překladač pro procesory TMS320C6xxx,
- `_TMS320C6400` – pokud je program překládán pro procesory TMS320C64xx,
- `_TMS320C6700` – pokud je program překládán pro procesory TMS320C67xx,

Další v dokumentaci překladače `spru187t`.

Využití předdefinovaných symbolů

- Kontrola spuštění správné verze

```
printf( "Program version from: %s %s\n", __DATE__, __TIME__);
```

- kontrola správné verze knihovny nebo procesoru

```
#ifndef __LITTLE_ENDIAN__
  #error( "Library needs little endian memory organization");
#endif
```

- trasovací výpisy

```
printf( "Variable x at %s:%d is equal to %d", __FILE__, __LINE__, x);
```

Makra preprocesoru

- Makra jsou během předzpracování zdrojových souborů rozvinuta a formální parametry jsou nahrazeny reálnými,
- `#define FLOAT2FIXED(x) ((short)((x)*32768 + 0.5))`
  `FLOAT2FIXED(0.635)`⇒`((short)((0.635)*32768 + 0.5))`
- je nutné uvádět závorky kolem formálních parametrů v definici makra,

```
#define MULT(x,y) x*y          MULT(2+5,3+4)⇒2+5*3+4
#define MULT(x,y) (x)*(y)      MULT(2+5,3+4)⇒(2+5)*(3+4)
```

- stejně tak je vhodné uvádět závorky kolem celého makra.

```
#define ADD(x,y) (x)+(y)       2*ADD(2,4)⇒2*(2)+(4)
#define ADD(x,y) ((x)+(y))     2*ADD(2,4)⇒2*((2)+(4))
```

Využití maker

- Jednoduché opakované operace

```
#define BITMASK(left,right)  (((1 << left)-1) & ~((1 << right)-1))
...

x = BITMASK(8,4);   // x = 0000000011110000
```

- trasovací výpisy

```
#define DPRINT(x) ((printf( "At %s:%d>", __FILE__, __LINE__)
```

- v případě problémů je možné se podívat na výstup preprocesoru s příponou `*.pp`,
- ponechání souborů se zapíná v nastavení projektu `Build->C6000 Compiler->Advanced Options->Parser Preprocessing Options`.

Sestavovací program

- Objektové soubory překladače a knihovny jsou tzv. *relocable* – adresy nejsou napevno, ale jsou použity jen symbolické názvy.
- Sestavovací program části spojí dohromady a symbolům přidělí výsledné adresy.
- Proces je řízen příkazy linkeru (soubor `*.cmd`).
- Výsledkem je spustitelný program a volitelně soubor `*.map` s přehledem obsazení paměti.

Více v dokumentaci `spru187t.pdf`.

Příklad souboru s příkazy linkeru

```
--rom_model
--heap_size=0x2000
--stack_size=0x0100
--library=rts6200.lib
MEMORY /* definice rozložení fyzické paměti */
{
  VECS:   o = 0x00000000     l = 0x000000400 /* reset & interrupt vectors     */
  PMEM:   o = 0x00000400     l = 0x00000FC00 /* intended for initialization   */
  BMEM:   o = 0x80000000     l = 0x000010000 /* .bss, .sysmem, .stack, .cinit */
}
SECTIONS /* přidělení fyzické paměti sekcím */
{
  vectors     >       VECS
  .text       >       PMEM
  .data       >       BMEM
  .stack      >       BMEM
  .bss        >       BMEM
  .sysmem     >       BMEM
  .cinit      >       BMEM
  .const      >       BMEM
  .cio        >       BMEM
  .far        >       BMEM
}
```

Předdefinované sekce

- `.text` – spustitelný kód a konstanty,
- `.stack` – zásobník,
- `.bss` – neinicializované globální a statické proměnné,
- `.sysmem` – halda (heap),
- `.cinit` – hodnoty pro explicitní inicializaci statických proměnných při ROM modelu,
- `.data` – hodnoty pro explicitní inicializaci statických proměnných při RAM modelu,
- `.const` – inicializované globální konstanty, statické proměnné, řetězcové konstanty,
- `.cio` – pro vstupně výstupní operace,
- `.far` – globální a statické proměnné adresované v režimu `far`.

Režimy překladu

- Režim `near`:
  - výchozí režim adresování proměnných pomocí posunutí vůči ukazateli zásobníku nebo haldy,
  - rychlý krátký program,
  - nemožnost posunutí o víc jak $2^{15} = 32\,\mathrm{KB}$ adres,
  - vynucení parametrem `--near` v nastavení sestavovacího programu.
- Režim `far`:
  - rozšířený režim adresování pomocí absolutní adres,
  - pomalý dlouhý program,
  - vynucení parametrem `--far` v nastavení sestavovacího programu,
  - klíčové slovo `far` u deklarace proměnné.

Režimy překladu

- Režim `ROM`:
  - výchozí režim inicializace proměnných,
  - předpokládá se, že není k dispozici paměť typu ROM,
  - konstanty pro inicializaci musí být součástí programu,
  - vynucení parametrem `--rom_model` v nastavení sestavovacího programu.
- Režim `RAM`:
  - rozšířený režim inicializace proměnných,
  - předpokládá se, že konstanty pro inicializaci jsou uloženy v paměti typu ROM,
  - nemusí tak být součástí programu,
  - vynucení parametrem `--ram_model` v nastavení sestavovacího programu.

Implementačně závislé příkazy

- Jazyk ANSI C počítal s použitím pro nejrůznější procesory, pro které bude nutné definovat nové přikazy závislé na vlastnostech použitého procesoru,
- definuje příkaz překladače #pragma, který umožňuje definovat nové příkazy platné pouze pro konkrétní překladač,
- např.

```
#pragma INTERRUPT  // obslužná funkce přerušení
#pragma DATA_ALIGN   // zarovnání dat na násobek
                     // nějaké hodnoty
#pragma DATA_SECTION // překlad bude zapsán do
                     // specifické sekce datové paměti
#pragma CODE_SECTION // překlad bude zapsán do
                     // specifické sekce paměti programu
#pragma MUST_ITERATE // definice minimálního počtu
                     // opakování smyčky
```

Definice dodatečných informací

Mnohem častěji je také nutné sdělit překladači, že některá proměnná

- je uložena přímo v registrech – klíčové slovo `register` – např. stavový register CSR,
- se může změnit asynchronně (např. hardwarově) klíčovým slovem `volatile`,
- že daný ukazatel je jediný možným přístupem k dané paměti (v podstatě opak `volatile`) – klíčové slovo `restrict`.

Výhody/nevýhody rozdělení programu do modulů

Výhody:

- Logicky související funkce jsou sloučeny do jednoho souboru,
- snadné rozdělení práce mezi více vývojářů,
- snadné znovu využití již hotových modulů – knihoven,

Nevýhody:

- Při překladu souboru jsou proměnné a funkce z jiných modulů neznámé – deklarovat v hlavičkovém souboru s klíčovým slovem `extern`,
- v různých modulech globální proměnné stejného názvu – klíčové slovo `static` omezí platnost pouze na aktuální soubor,
- názvy exportovaných funkcí je vhodné doplnit předponou názvu modulu – např. `DSK6416_LED_on`, `DSK6713_LED_on`.

Deklarace vs. definice vs. inicializace

- Deklarace – jenom informace o tom, že někde je definován objekt (proměnná, pole, struktura,funkce, atd.) s danými vlastnostmi,
```
void funkce( int, short);
#extern int pole[];
```
- Definice – vyhrazení místa v paměti pro daný objekt.
```
int pole[];
void funkce(int a, short b){
}
```
- Inicializace – naplnění vyhrazeného místa v paměti daným obsahem.
```
int pole[] = { 1, 2, 3, 4};
```

V hlavičkových souborech používat pouze deklarace.

Datové typy

- Jazyk ANSI C nezná pojem zlomkové číslo,
- rozeznává pouze celočíselné proměnné – `short`, `int`, `long` – a reálné – `float`, `double`,
- překladač Code Composer Studio využívá toho, že ANSI C nemá striktně definován počet bitů (jen `long` $\geq$ `int` $\geq$ `short`), a přizpůsobuje počet bitů datových typů délce registrů (`short` – 16 bitů, `int` – 32 bitů, `long` – 40 bitů) POZOR: většina překladačů předpokládá sizeof(int) = sizeof(long) = 32 bitů,
- rozlišení celý/zlomkový datový typ se pak děje podle prováděné operace,
- některé překladače (CodeWarrior) definují nové datové typy pro zlomková čísla podle délky registrů:
  __fixed__ – 16 bitů, __shortfixed__ – 16 bitů, __longfixed__ – 32 bitů.

Použití instrukcí asembleru

- Většina překladačů podporuje vložení platné instrukce asembleru do zdrojového textu jazyka C pomocí funkce `asm`,
  - `asm( "STR: .byteäbc");`
  - nevýhodou tohoto přístupu je obtížná výměna dat mezi proměnnými definovanými v asembleru a proměnnými definovanými v jazyce C.
- další možnost je použití speciálních vnitřních funkcí překladače – intrinsic
  - často jsou deklarovány v jazyce C (ve zvláštním hlavičkovém souboru) a používají se jako jakákoliv funkce jazyka C,
  - překládají se většinou jako jediná instrukce procesoru,
  - výhodou je možnost používat přímo proměnné jazyka C,
  - např.

```
int _smpy( int, int);
int _mpy( int, int);
int _smpyh( int, int);
```

- volání celých funkcí v asembleru z jazyka C.

Příklady intrinsic funkcí TMS320C000

instrukce	prototyp	význam
EXT	`int _ext( int x, unsigned a, unsigned b)`	extrakce části bitů ((x«a)»b) se znaménkem
EXTU	`int _ext( int x, unsigned a, unsigned b)`	extrakce části bitů ((x«a)»b) bez znaménka
NORM NORM	`unsigned _norm( int x)` `unsigned _lnorm( long x)`	počet redundantních znaménkových bitů
SADD SADD	`int _sadd( int x, int y)` `int _lsadd( int x, long y)`	součet (x + y) se saturací
SAT	`int _sat( long x)`	saturace 40bitové hodnoty na 32bitovou

SMPY
SMPYH
SMPYHL
SMPYLH
	`int _smpy( int x, int y)`
	`int _smpyh( int x, int y)`
	`int _smpyhl( int x, int y)`
	`int _smpylh( int x, int y)`
		násobení s bitovým posunem vlevo a saturací (násobení zlomkových čísel)

SSHL	`int _sshl( int x, unsigned a)`	bitový posun vlevo (x « a) se saturací
SSUB SSUB	`int _ssub( int x, int y)` `int _lssub( int x, long y)`	rozdíl (x - y) se saturací