Cm VM

—

for

Win32 Visual Studio Community 2019

Michel de Champlain
Department of Computer Science and Software Engineering
Concordia University, Montreal, Canada

November 9, 2020

Contents

Chapter 1

BSL and HAL

1.1 Board Support Layer (Machine Dependent): Cout Lib

1.1.1 _console.c

```
1   /* _console.c -- Console Interface for Win32 (with Microsoft Visual C/C++ 2015 - VS14) which isolates 'putchar()'
2   //               to avoid including <stdio.h> bringing all sort of conflicts with its kitchen sick of macros.
3   //
4   // Copyright (C) 1985-2020 by Michel de Champlain
5   //
6   // Jun 24, 2014 - just to wrap putchar() and avoid including <stdio.h> anywhere else.
7   */
8
9   #include <stdio.h>    /* for only: putchar, getchar, fflush, stdin */
10
11  void Console_Putchar(char c) { putchar(c); }
```

1.1.2 _cout.c

```
1   /* _cout.c - Implementation of a Console for Cm Hardware Abstract Layer for Output Interface.
2   //
3   // Copyright (C) 1999-2020 by Michel de Champlain
4   //
5   */
6
7   #include "_outdesc.h"
8
9   #if ConsoleOutputWithPrintf
10
11  #include <stdio.h>
12
13  static void COut_PutB(bool b)        { printf("%s", b ? "true" : "false"); }
14  static void COut_PutC(char c)        { printf("%c", c); }
15  static void COut_PutS(const char* s) { printf("%s", s); }
16  static void COut_PutI(i32  i)        { printf("%ld", i); }
17  static void COut_PutU(u32  u)        { printf("%lu", u); }
18  static void COut_PutX(u32  u)        { printf("%08lX", u); } // To make hex output always aligned to 8 hex digits.
19  static void COut_PutN(void)          { printf("\n"); }
20
21  #else
22  #include "_xtoa.h"
23
24  // External refs to 'console.c' without
25  void  Console_Putchar(char  c);
26
27  static char  buf[12];               /* to cover max size (12) "i32" (10+sign+null) */
28
29  static void COut_PutB(bool b)        { Console_Putchar(b ? 'T' : 'F'); }
30  static void COut_PutC(char c)        { Console_Putchar(c); }
```

```
31  static void COut_PutS(const char* s) { while (*s) Console_Putchar(*s++); }
32  static void COut_PutI(i32  i)        { System_itoa(i, buf); COut_PutS(buf); }
33  static void COut_PutU(u32  u)        { System_utoa(u, buf, 10); COut_PutS(buf); }
34  static void COut_PutX(u32  x)        { System_utoa(x, buf, 16); COut_PutS(buf); } // Same behavior as Dos16 VM:
35                                                                                    // Hex alpha in upppercase
36  static void COut_PutN(void)          { Console_Putchar('\n'); }
37  #endif
38
39  static IVMOutDesc cout = {
40      COut_PutB,
41      COut_PutC,
42      COut_PutI,
43      COut_PutU,
44      COut_PutS,
45      COut_PutX,
46      COut_PutN
47  };
48
49  IOut Out_GetFactory(const char* whichOne) {
50      whichOne = 0; // To avoid the warning on the unreferenced formal parameter
51      return &cout;
52  }
```

1.1.3 _out.h

```
1   /* _out.h - Interface for Cm VM Output Interface
2   //
3   // Copyright (C) 1999-2020 by Michel de Champlain
4   //
5   */
6
7   #ifndef __CmVM_Out_h
8   #define __CmVM_Out_h
9
10  #include "_stdtype.h"
11
12          struct IVMOutDesc;
13  typedef struct IVMOutDesc* IOut;
14
15  IOut Out_GetFactory(const char* whichOne);
16
17  #endif
```

1.1.4 _outdesc.h

```
1   /* _outdesc.h - Interface for VM Output Descriptor
2   //
3   // Copyright (C) 1999-2020 by Michel de Champlain
4   //
5   */
6
7   #ifndef __CmVM_OutDesc_h
8   #define __CmVM_OutDesc_h
9
10  #include "_out.h"
11
12  // Private VM Output Function Pointer Types:
13  typedef void (*VMPutB)(bool);
14  typedef void (*VMPutC)(char);
15  typedef void (*VMPutI)(i32);
16  typedef void (*VMPutU)(u32);
17  typedef void (*VMPutS)(const char*);
18  typedef void (*VMPutX)(u32);
19  typedef void (*VMPutN)(void);
20
21  // Private Interface Output Descriptor
22  typedef struct IVMOutDesc {
```

```
23      VMPutB pb;
24      VMPutC pc;
25      VMPutI pi;
26      VMPutU pu;
27      VMPutS ps;
28      VMPutX px;
29      VMPutN pn;
30  } IVMOutDesc;
31
32  #endif
```

1.1.5 _stdtype.h

```
1   /* _stdtype.h - Cm VM Standard (basic) type definitions (VS2019 Host Version)
2   //
3   // Copyright (C) 1999-2020 by Michel de Champlain
4   //
5   */
6
7   #ifndef __CmVM_stdtype_h
8   #define __CmVM_stdtype_h
9
10  #include <stdint.h>
11
12  #define bool               unsigned long
13  #define false              ((bool)0)
14  #define true               ((bool)1)
15
16  typedef uint8_t   u8;
17  typedef int8_t    i8;
18  typedef uint16_t  u16;
19  typedef int16_t   i16;
20  typedef uint32_t  u32;
21  typedef int32_t   i32;
22  typedef float     f32;
23
24  #ifdef Ptr16bits
25  typedef           u16*     ptr;
26  #else
27  typedef           u32*     ptr;
28  #endif
29
30  #endif
```

1.1.6 _xtoa.h

```
1   /* _xtoa.h - 'to ascii' functions (xtoa): itoa, utoa, and ftoa (used only as private functions)
2   //
3   // Copyright (C) 1999-2020 by Michel de Champlain
4   //
5   */
6
7   #ifndef __CmVM_xtoa_h
8   #define __CmVM_xtoa_h
9
10  #include "_stdtype.h"
11
12  void _utoa(u32 n, char* buf, int next, u8 base);
13
14  void System_itoa(i32 i, char* buf);
15
16  #define     System_utoa(n,buf,base)          _utoa(n,buf,0,base)
17
18  #endif
```

1.1.7 _xtoa.c

```
1    /* _xtoa.c - 'to ascii' functions (xtoa): itoa, utoa, and ftoa (used only as private functions)
2    //
3    // Copyright (C) 1999-2020 by Michel de Champlain
4    //
5    */
6
7    #include "_xtoa.h"
8
9    /* Converts 32-bit unsigned integer to a buffer. Base is 16 by default. */
10   void _utoa(u32 n, char* buf, int next, u8 base) {
11       u32  r, f;
12       bool foundNonzero = false;
13
14       if (base == 10)
15           f = 1000000000L;
16       else if (base == 16)
17           f = 0x10000000L;
18       else {
19           f = 0x10000000L;
20           base = 16;
21       }
22
23       if (n == 0) {
24           buf[next++] = '0';
25       } else {
26           while (f > 0) {
27               r = n / f;
28               if (foundNonzero || r > 0) {
29                   if (base == 10)
30                       buf[next++] = (char)(r+'0');
31                   else
32                       buf[next++] = (char)(r >= 10 ? r-10+'A': r+'0');
33                   foundNonzero = true;
34               }
35               n -= r * f;
36               f /= base;
37           }
38       }
39       buf[next] = '\0';
40   }
41
42   /* Converts 32-bit signed integer to a buffer. Base is 10 by default. */
43   void System_itoa(i32 i, char* buf) {
44       int  next = 0;
45       if (i < 0L) {
46           buf[next++] = '-';
47           i = -i;
48       }
49       _utoa(i, buf, next, 10);
50   }
```

1.2 Hardware Abstract Layer

1.2.1 hal.h

```
1    /* hal.h -- Hardware Abstraction Layer interface which decouples (or bridges)
2    //          the board support (machine dependent) modules to the VM portable code.
3    //
4    // Copyright (C) 1985-2020 by Michel de Champlain
5    //
6    */
7
8    #ifndef __hal_h
9    #define __hal_h
10
```

```
11   #include "_out.h"
12
13   void Hal_Init(void);
14
15   #endif
```

1.2.2 out.h

```
1    /* out.h - Cm VM Console Out Interface - no more macros.
2    //
3    // Copyright (C) 1999-2020 by Michel de Champlain
4    //
5    */
6
7    #ifndef __CmVM_VMOut_h
8    #define __CmVM_VMOut_h
9
10   #include "_out.h"
11
12   void VMOut_Init(IOut out);
13
14   void VMOut_PutB(bool b);
15   void VMOut_PutC(char c);
16   void VMOut_PutI(i32  i);
17   void VMOut_PutU(u32  u);
18   void VMOut_PutS(const char* s);
19   void VMOut_PutX(u32  x);
20   void VMOut_PutN(void);
21
22   #endif
```

1.2.3 hal.c

```
1    /* hal.c -- Hardware Abstraction Layer implementation
2    //
3    // Copyright (C) 1985-2020 by Michel de Champlain
4    //
5    */
6
7    #include "hal.h"
8    #include "out.h"
9
10   void Hal_Init(void) {
11       VMOut_Init(Out_GetFactory("")); // "" to save space, later should be "console".
12
13   #ifdef FullVersion
14       Add other init subsystems here.
15   #endif
16   }
```

1.2.4 out.c

```
1    /* out.c - Out on Console for the VM Host
2    //
3    // Copyright (C) 1999-2020 by Michel de Champlain
4    //
5    */
6
7    #include "_outdesc.h"
8    #include "out.h"
9
10   static IOut vmOut;
11
12   void VMOut_Init(IOut out)      { vmOut = out;  }
13
14   void VMOut_PutB(bool b)        { vmOut->pb(b); }
```

```
15   void VMOut_PutC(char c)        { vmOut->pc(c); }
16   void VMOut_PutI(i32  i)        { vmOut->pi(i); }
17   void VMOut_PutU(u32  u)        { vmOut->pu(u); }
18   void VMOut_PutS(const char* s) { vmOut->ps(s); }
19   void VMOut_PutX(u32  x)        { vmOut->px(x); }
20   void VMOut_PutN(void)          { vmOut->pn();  }
```

Chapter 2

VM Operand Stack

2.1 vmstack.h

```
1   /* vmstack.h - Cm VM Operand Stack - simplified to have one instance only (no threads).
2   //
3   // Copyright (C) 1999-2020 by Michel de Champlain
4   //
5   */
6
7   #ifndef __CmVMStack_h
8   #define __CmVMStack_h
9
10  #include "_stdtype.h"
11
12          struct VMStackDesc;
13  typedef struct VMStackDesc*  Stack;
14
15  Stack     Stack_New     (u8 initialCapacity);
16  void      Stack_Delete  (Stack s);
17  i32       Stack_Count   (Stack s);
18  u8        Stack_Capacity(Stack s);
19  void      Stack_Push    (Stack s, i32 item);
20  i32       Stack_Pop     (Stack s);
21
22          struct StackEnumtorDesc;
23  typedef struct StackEnumtorDesc*  StackEnumtor;
24
25  StackEnumtor Stack_GetEnumerator(Stack s);
26
27  void     StackEnumtor_Delete  (StackEnumtor s);
28  bool     StackEnumtor_MoveNext(StackEnumtor s);
29  u32      StackEnumtor_Current (StackEnumtor s);
30  void     StackEnumtor_Reset   (StackEnumtor s);
31  //------------------------------------------------- VMSTACK_BASIC
32
33  bool   Stack_IsEmpty(Stack s);
34
35  #ifdef STACK_CHECK
36  i8     Stack_WaterMark(Stack s);
37  #endif
38
39  StackEnumtor Stack_GetEnumerator(Stack s);
40
41  void   Stack_Print  (StackEnumtor e);
42
43  // ----------------------------------- VMSTACK_INHERENT Instruction
44
45  i32    Stack_top(Stack s);
46  void   Stack_dup(Stack s);
47  void   Stack_not(Stack s);
```

```
48  void   Stack_and(Stack s);
49  void   Stack_or (Stack s);
50  void   Stack_xor(Stack s);
51  void   Stack_neg(Stack s);
52  void   Stack_inc(Stack s);
53  void   Stack_dec(Stack s);
54  void   Stack_add(Stack s);
55  void   Stack_sub(Stack s);
56  void   Stack_mul(Stack s);
57  void   Stack_div(Stack s);
58  void   Stack_rem(Stack s);
59  void   Stack_shl(Stack s);
60  void   Stack_shr(Stack s);
61  void   Stack_teq(Stack s);
62  void   Stack_tne(Stack s);
63  void   Stack_tlt(Stack s);
64  void   Stack_tgt(Stack s);
65  void   Stack_tle(Stack s);
66  void   Stack_tge(Stack s);
67
68  void   Stack_enterU5(Stack s, u8 funcInfo);
69  void   Stack_enterU8(Stack s, u8 funcInfo);
70  void   Stack_exit   (Stack s);
71
72  void   Stack_addVariable(Stack s, u8 var);
73  void   Stack_loadVariable (Stack s, u8 var);
74  void   Stack_storeVariable(Stack s, u8 var);
75
76  #endif // __CmVMStack_h
```

2.2 vmstack.c

```
1   /* vmstack.c - Cm VM Operand Stack - simplified to have one instance only (no threads).
2   //
3   // Copyright (C) 1999-2020 by Michel de Champlain
4   //
5   */
6
7   #include "out.h"
8   #include "vmstack.h"
9
10  typedef struct VMStackDesc {
11      i32         si[32];    // Simplification for Cm VM - one instance only.
12      i8          bp;        /* base (or frame) pointer */
13      i8          sp;        /* stack pointer (sp+1 == number of elements) : to avoid size ptr calculation (sp-base) */
14      u8          capacity;  /* maximum number of elements: to avoid overflow calculation (limit-base) and sp/limit */
15  #ifdef STACK_CHECK
16      i8          waterMark; /* used to estimate the effective stack space used for a specific thread */
17  #endif
18  } VMStackDesc;
19
20  #ifdef STACK_CHECK
21  static void printMsg(Stack s, char* msg) {
22          VMOut_PutS("\nStack: "); VMOut_PutS(msg);
23          VMOut_PutS(" ** limit["); VMOut_PutU( Stack_Capacity(s) );
24          VMOut_PutS("] used(watermark)["); VMOut_PutI((i32)s->waterMark);
25          VMOut_PutS("] sp["); VMOut_PutI((i32)s->sp);
26          VMOut_PutS("] bp["); VMOut_PutI((i32)s->bp);
27          VMOut_PutS("]\n");
28  }
29  #endif
30
31  static VMStackDesc vmStackDesc; // One instance.
32
33  Stack Stack_New(u8 capacity) {
34      Stack  s = &vmStackDesc;    // One instance.
```

```
35      if (s   &&   capacity != 0) {
36          s->bp = s->sp = -1;
37          s->capacity = capacity;
38  #ifdef STACK_CHECK
39          s->limit = s->stack + capacity;
40          s->waterMark = -1;
41          printMsg(s, "init");
42  #endif
43      }
44      return s;
45  }
46
47  void Stack_Delete(Stack s)   { } // One instance.
48
49  i32  Stack_Count   (Stack s) { return s->sp+1;     }
50  u8   Stack_Capacity(Stack s) { return s->capacity; }
51
52  void Stack_Push(Stack s, i32 value) {
53  //t    VMOut_PutS("\nsp = "); VMOut_PutI((i32)s->sp); VMOut_PutS(", capacity = "); VMOut_PutI((i32)s->capacity); VMOut_PutN
54  #ifdef STACK_CHECK
55      if (s->sp >= Stack_Capacity(s)-1) { printMsg(s, "overflow"); return; }
56  #endif
57      s->si[++s->sp] = value;
58  #ifdef STACK_CHECK
59      if (s->sp > s->waterMark) {
60          s->waterMark = s->sp;
61      }
62      VMOut_PutS("\nsp = "); VMOut_PutI((i32)s->sp); VMOut_PutS(", waterMark = "); VMOut_PutI((i32)s->waterMark); VMOut_PutN(
63  #endif
64  //t    VMOut_PutS("\nPush: val = "); VMOut_PutI(value); VMOut_PutN();
65  }
66
67  i32 Stack_Pop(Stack s) {
68  #ifdef STACK_CHECK
69      if (s->sp < 0)
70          printMsg(s, "underflow");
71  //    else
72  #endif
73      return s->si[s->sp--];
74  }
75
76  bool Stack_IsEmpty(Stack s) { return s->sp == -1; }
77
78  #ifdef STACK_CHECK
79  i8 Stack_WaterMark(Stack s) { return s->waterMark; }
80  #endif
81
82  typedef struct StackEnumtorDesc {
83      Stack  stack;
84      i8     index;
85  } StackEnumtorDesc;
86
87  static StackEnumtorDesc stackEnumtorDesc; // One instance.
88
89  StackEnumtor StackEnumtor_New(Stack s) {
90      StackEnumtor e = &stackEnumtorDesc;   // One instance.
91      e->stack = s;
92      StackEnumtor_Reset(e);
93      return e;
94  }
95  StackEnumtor Stack_GetEnumerator(Stack s) {
96      return StackEnumtor_New(s);
97  }
98  void StackEnumtor_Delete(StackEnumtor s) {  } // One instance.
99
100 bool StackEnumtor_MoveNext(StackEnumtor s) { return --s->index >= 0; }
101 u32  StackEnumtor_Current(StackEnumtor s)  { return (u32)s->stack->si[s->index]; }
102 void StackEnumtor_Reset(StackEnumtor s)    { s->index = s->stack->sp+1; }
```

```
103
104 void Stack_Print(StackEnumtor e) {
105     StackEnumtor_Reset(e);
106     VMOut_PutC('[');
107     while ( StackEnumtor_MoveNext(e) ) {
108         u32 n = StackEnumtor_Current(e);
109         VMOut_PutS(" ");
110         VMOut_PutX(n);
111     }
112     VMOut_PutS(" ]");
113 }
114
115 // ---------------------- Inherent Mode Support
116
117 i32  Stack_top (Stack s) { return s->si[s->sp]; }
118 void Stack_dup(Stack s)  { Stack_Push(s, s->si[s->sp]); }
119 void Stack_not(Stack s)  { s->si[s->sp] = ~s->si[s->sp]; }
120
121 void Stack_and(Stack s) {
122     i32 v2 = s->si[s->sp--]; /* pop */
123     i32 v1 = s->si[s->sp];
124     s->si[s->sp] = (i32)(v1 & v2);
125 }
126 void Stack_or(Stack s) {
127     i32 v2 = s->si[s->sp--]; /* pop */
128     i32 v1 = s->si[s->sp];
129     s->si[s->sp] = (i32)(v1 | v2);
130 }
131 void Stack_xor(Stack s) {
132     i32 v2 = s->si[s->sp--]; /* pop */
133     i32 v1 = s->si[s->sp];
134     s->si[s->sp] = (i32)(v1 ^ v2);
135 }
136 void Stack_neg(Stack s) { s->si[s->sp] = -s->si[s->sp]; }
137 void Stack_inc(Stack s) { s->si[s->sp]++; } /* ++*s->sp; */
138 void Stack_dec(Stack s) { s->si[s->sp]--; } /* --*s->sp; */
139 void Stack_add(Stack s) {
140     i32 v2 = s->si[s->sp--]; /* pop */
141     s->si[s->sp] += v2;
142 }
143 void Stack_sub(Stack s) {
144     i32 v2 = s->si[s->sp--]; /* pop */
145     s->si[s->sp] -= v2;
146 }
147 void Stack_mul(Stack s) {
148     i32 v2 = s->si[s->sp--]; /* pop */
149     s->si[s->sp] *= v2;
150 }
151 void Stack_div(Stack s) {
152     i32 v2 = s->si[s->sp--]; /* pop */
153     if (v2 == 0) {
154         VMOut_PutS("Division by zero\n");
155         return;
156     }
157     s->si[s->sp] /= v2;
158 }
159 void Stack_rem(Stack s) {
160     i32 v2 = s->si[s->sp--]; /* pop */
161     s->si[s->sp] %= v2;
162 }
163 void Stack_shl(Stack s) {
164     i32 v2 = s->si[s->sp--]; /* pop */
165     s->si[s->sp] <<= v2;
166 }
167 void Stack_shr(Stack s) {
168     i32 v2 = s->si[s->sp--]; /* pop */
169     s->si[s->sp] >>= v2;
170 }
```

```
171 void Stack_teq(Stack s) {
172     i32 v2 = s->si[s->sp--]; /* pop */
173     i32 v1 = s->si[s->sp];
174     s->si[s->sp] = (v1 == v2)? 1 : 0;
175 }
176 void Stack_tne(Stack s) {
177     i32 v2 = s->si[s->sp--]; /* pop */
178     i32 v1 = s->si[s->sp];
179     s->si[s->sp] = (v1 != v2)? 1 : 0;
180 }
181 void Stack_tlt(Stack s) {
182     i32 v2 = s->si[s->sp--]; /* pop */
183     i32 v1 = s->si[s->sp];
184     s->si[s->sp] = (v1 < v2);
185 }
186 void Stack_tgt(Stack s) {
187     i32 v2 = s->si[s->sp--]; /* pop */
188     i32 v1 = s->si[s->sp];
189     s->si[s->sp] = (v1 > v2);
190 }
191 void Stack_tle(Stack s) {
192     i32 v2 = s->si[s->sp--]; /* pop */
193     i32 v1 = s->si[s->sp];
194     s->si[s->sp] = (v1 <= v2);
195 }
196 void Stack_tge(Stack s) {
197     i32 v2 = s->si[s->sp--]; /* pop */
198     i32 v1 = s->si[s->sp];
199     s->si[s->sp] = (v1 >= v2);
200 }
201 //------------------------------------------- Support for Functions
202 void Stack_enterU5(Stack s, u8 funcInfo) {
203 //t    VMOut_PutS("enter.u5: ENTRY sp = "); VMOut_PutX((u32)s->sp); VMOut_PutS("; bp = "); VMOut_PutX((u32)s->bp); VMOut_Pu
204     u8  v  = (funcInfo >> 4) & 0x01;
205     u8  np = (funcInfo >> 2) & 0x03;
206     u8  nl =  funcInfo       & 0x03;
207     u8  fi = (v << 6) | (np << 3) | nl;
208     i32 retAddr = s->si[s->sp--];   /* pop (save) caller's return address    */
209 #ifdef MONITOR
210    VMOut_PutS("enter.u5: v = "); VMOut_PutU((u32)v);
211    VMOut_PutS("; np = ");     VMOut_PutX((u32)np);  VMOut_PutS("; nl = ");     VMOut_PutX((u32)nl);
212    VMOut_PutS("; fi = ");     VMOut_PutX((u32)fi);  VMOut_PutS("; oldbp = ");  VMOut_PutI((u32)s->bp); VMOut_PutN();
213 #endif
214     s->sp  += nl;                /* allocate space for local variables         */
215     s->si[++s->sp] = fi;       /* push function information                  */
216     s->si[++s->sp] = retAddr;  /* push back the caller's return address      */
217     s->si[++s->sp] = s->bp; /* push (save) caller's bp (frame context)    */
218     s->bp = s->sp;             /* set frame context for the current function */
219 //t    VMOut_PutS("enter.u5: EXIT  sp = "); VMOut_PutX((u32)s->sp); VMOut_PutS("; bp = "); VMOut_PutX((u32)s->bp); VMOut_Pu
220 }
221 void Stack_enterU8(Stack s, u8 funcInfo) {
222     u8  v  = (funcInfo >> 6) & 0x01;
223     u8  np = (funcInfo >> 3) & 0x07;
224     u8  nl =  funcInfo       & 0x07;
225     u8  fi = (v << 6) | (np << 3) | nl;
226     i32 retAddr = s->si[s->sp--];   /* pop (save) caller's return address    */
227 #ifdef MONITOR
228    VMOut_PutS("enter.u8: v = "); VMOut_PutU((u32)v);
229    VMOut_PutS("; np = ");        VMOut_PutX((u32)np);  VMOut_PutS("; nl = ");     VMOut_PutX((u32)nl);
230    VMOut_PutS("; fi = ");        VMOut_PutX((u32)fi);  VMOut_PutS("; oldbp = ");  VMOut_PutI((u32)s->bp); VMOut_PutN();
231 #endif
232     s->sp  += nl;                /* allocate space for local variables         */
233     s->si[++s->sp] = fi;       /* push function information                  */
234     s->si[++s->sp] = retAddr;  /* push back the caller's return address      */
235     s->si[++s->sp] = s->bp; /* push (save) caller's bp (frame context)    */
236     s->bp    = s->sp;          /* set frame context for the current function */
237 }
238 void Stack_exit(Stack s) {
```

```
239 //t    VMOut_PutS("exit: ENTRY sp = "); VMOut_PutX((u32)s->sp); VMOut_PutS("; bp = "); VMOut_PutX((u32)s->bp); VMOut_PutN()
240     u8   fi = (u8)s->si[s->bp-2];
241     bool v  = (fi >> 6) & 0x01;
242     u8   np = (fi >> 3) & 0x07;
243     u8   nl =  fi       & 0x07;
244     i32   retAddr, retVal = 0L; // to avoid warning not initialized.
245 #ifdef MONITOR
246    VMOut_PutS("exit:  v = "); VMOut_PutU((u32)v);
247    VMOut_PutS("; np = ");     VMOut_PutX((u32)np);  VMOut_PutS("; nl = ");     VMOut_PutX((u32)nl);
248    VMOut_PutS("; fi = ");     VMOut_PutX((u32)fi);  VMOut_PutS("; oldbp = ");  VMOut_PutI((i32)s->bp); VMOut_PutN();
249 #endif
250     if (v) retVal = s->si[s->sp--];       /* save the return value (if any)            */
251     s->bp      = (u8)s->si[s->sp--];   /* pop (restore) caller's bp (frame context) */
252     retAddr       = s->si[s->sp--];       /* pop (save) caller's return address        */
253     s->sp     -= (np+nl+1);                  /* deallocate space for parameters, local
254                                                    variables, and function information       */
255     if (np+nl > 7) s->sp -= nl; /* March 8 */
256
257     if (v) s->si[++s->sp] = retVal;       /* push back the return value (if any)       */
258     s->si[++s->sp] = retAddr;             /* push back the caller's return address     */
259 //t    VMOut_PutS("exit: EXIT  sp = "); VMOut_PutX((u32)s->sp); VMOut_PutS("; bp = "); VMOut_PutX((u32)s->bp); VMOut_PutN()
260 }
261
262 static u8 getFrameOffset(u8 v, u8 np, u8 nl) {
263     if (np+nl > 7)
264         return (u8)2 + np + nl - v + nl;
265     else
266         return (u8)2 + np + nl - v;
267 }
268 //-----------------------------------------------------------------------------
269 void Stack_addVariable(Stack s, u8 var) {
270     u8 fi = (u8)s->si[s->bp-2];
271     u8 np = (fi >> 3) & 0x07;
272     u8 nl =  fi       & 0x07;
273     u8 fo = getFrameOffset(var, np, nl);
274 #ifdef MONITOR
275    VMOut_PutS("addv: var = "); VMOut_PutU((u32)var);
276    VMOut_PutS("; fi = ");      VMOut_PutX((u32)fi);
277    VMOut_PutS("; fo = ");      VMOut_PutX((u32)fo);
278    VMOut_PutS("; v = ");       VMOut_PutX((u32)s->si[s->bp-fo].data.ival); VMOut_PutN();
279 #endif
280     s->si[s->bp-fo] += Stack_Pop(s);
281 }
282 //-----------------------------------------------------------------------------
283 void Stack_loadVariable(Stack s, u8 var) {
284     u8 fi = (u8)s->si[s->bp-2];
285     u8 np = (fi >> 3) & 0x07;
286     u8 nl =  fi       & 0x07;
287     u8 fo = getFrameOffset(var, np, nl);
288 #ifdef MONITOR
289    VMOut_PutS("\r\n");
290    VMOut_PutS("ldv: var = "); VMOut_PutU((u32)var);
291    VMOut_PutS("; fi = ");     VMOut_PutX((u32)fi);
292    VMOut_PutS("; fo = ");     VMOut_PutX((u32)fo);
293    VMOut_PutS("; v = ");      VMOut_PutX((u32)s->si[s->bp-fo]); VMOut_PutN();
294 #endif
295     Stack_Push(s, s->si[s->bp-fo]);
296 }
297 //-----------------------------------------------------------------------------
298 void Stack_storeVariable(Stack s, u8 var) {
299     u8  fi = (u8)s->si[s->bp-2];
300     u8  np = (fi >> 3) & 0x07;
301     u8  nl =  fi       & 0x07;
302     u8  fo = getFrameOffset(var, np, nl);
303 #ifdef MONITOR
304    VMOut_PutS("\r\n");
305    VMOut_PutS("stv: var = "); VMOut_PutU((u32)var);
306    VMOut_PutS("; fi = ");     VMOut_PutX((u32)fi);
```

```
307     VMOut_PutS("; fo = ");      VMOut_PutX((u32)fo);
308     VMOut_PutS("; v = ");       VMOut_PutX((u32)s->si[s->bp-fo]); VMOut_PutN();
309  #endif
310      s->si[s->bp-fo] = Stack_Pop(s);
311  }
```

Chapter 3

VM Admin and Core

3.1 admin.c

```
1   /* admin.c - admin for the Cm Embedded Virtual Machine which:
2   //         - isolates the <stdio.h> with all put* in the VM
3   //         - defines _CRT_SECURE_NO_WARNINGS to avoid all MS secure crap on **_s
4   //
5   // Copyright (C) 1999-2020 by Michel de Champlain
6   //
7   */
8
9   #include <stdio.h>  /* for FILE   */
10  #include <string.h> /* for strtok */
11
12  #include "hal.h"
13  #include "out.h"
14  #include "vm.h"
15
16  #ifdef Dos16
17  #define Target      "(Dos16)"
18  #elif defined(Arm7)
19  #define Target      "(Arm7)"
20  #else
21  #define Target      "(Win32)"
22  #endif
23
24  #if LaterForSoen422SerialLoader
25  #include "IStream.h"
26  #include "Stream.h"
27
28  #include "ILoader.h"
29  #include "Loader.h"
30  #endif
31
32  #define VMName      "Cm Virtual Machine "
33  #define AppSuffix   ""
34  #define AppName     "cm"
35  #define Version     " v0.1.00.1101a "
36  #define Copyright   "Copyright (c) 2001-2020  Michel de Champlain"
37
38  // Banner = VMname AppSuffix Version Copyright
39  static void DisplayBanner() {
40      VMOut_PutS(VMName); VMOut_PutS(AppSuffix); VMOut_PutS(Version); VMOut_PutS(Target); VMOut_PutN();
41      VMOut_PutS(Copyright); VMOut_PutN();
42  }
43  static void Usage() {
44      VMOut_PutS("\nUsage: "); VMOut_PutS(AppName); VMOut_PutS(" Options? <file.exe>\n");
45      VMOut_PutS("\n         -v          Display the version and exit.");
46      VMOut_PutS("\n         -? -help    Display options and exit.\n");
47  }
```

```
48
49  #define MemMax        4096
50  #define MemAllocated  (4096+1024)
51  /*public*/  u8*    mem;
52  /*public*/  u8     memAllocated[MemAllocated];
53
54  // To get the base RAM address on a memory segment increment.
55  static u8* GetBaseAddr(u8* memAddr, u32 memInc) {
56      u32 a = (u32)memAddr + memInc;
57      u32 m = memInc - 1U;
58  //t    VMOut_PutS("Admin: a = "); VMOut_PutX((u32)a); VMOut_PutN();
59  //t    VMOut_PutS("Admin: m = "); VMOut_PutX((u32)m); VMOut_PutN();
60
61      u32 r = a & ~m;
62  //t    VMOut_PutS("Admin: r = "); VMOut_PutX((u32)r); VMOut_PutN();
63      return (u8*)r;
64  }
65
66  FILE* file;
67
68  /* 1st two bytes are the size (msb:lsb) */
69  static bool loadObjFile(FILE* f, u16 maxSize) {
70      u16 n, size;
71      u8  buf[2];
72
73      buf[0] = (u8)fgetc(f);             // Read size.msb
74      buf[1] = (u8)fgetc(f);             // Read size.msb
75      size = (u16)((buf[0] << 8) | buf[1]);
76
77  //t VMOut_PutS("loadObjFile of size = %u\n", (u32)size);
78
79      if (size <= maxSize) {
80          for (n = 0; n < size; n++) {
81              mem[n] = (u8)fgetc(f);
82  #ifdef MONITOR
83              VMOut_PutS(".");
84              VMOut_PutS("%02x ", (u8)mem[n]);
85  #endif
86          }
87      } else {
88          VMOut_PutS("Executable file too big (should be <= "); VMOut_PutU((u32)maxSize); VMOut_PutS(" bytes).\n");
89          return false;
90      }
91      fclose(f);
92  #ifdef MONITOR
93      System_putc('\n'); System_putu(size); System_puts(" bytes loaded.\n");
94  #endif
95      return true;
96  }
97
98  // Returns the filename extention.
99  const char *GetFilenameExt(const char *filename) {
100     const char *dot = strrchr(filename, '.');
101     if(!dot || dot == filename) return "";
102     return dot + 1;
103 }
104
105 // Returns filename portion of the given path (for Unix or Windows)
106 // Returns empty string if path is directory
107 const char *GetFileName(const char *path) {
108     const char *pfile = path + strlen(path);
109     for (; pfile > path; pfile--) {
110         if ((*pfile == '\\') || (*pfile == '/')) {
111             pfile++;
112             break;
113         }
114     }
115     return pfile;
```

```
116 }
117
118 int main(int argc, char* argv[]) {
119     char  filename[200]; // On Win32, you need a big buffer because in VS IDE filenames are passed with full path.
120     const char* name;
121     const char* ext;
122     int   i = 1;
123
124 //t VMOut_PutS("argv[0] = [%s]\n", argv[0]);
125 //t VMOut_PutS("argv[1] = [%s]\n", argv[1]);
126
127     // Do Hal_Init() before any option messages.
128     Hal_Init();
129
130     // ********* Important to adjust memory before loading the file in memory.
131 //t    VMOut_PutS("GetBaseAddr(): sizeof u8* = "); VMOut_PutI((i32)sizeof(u8*)); VMOut_PutN();
132 //t    VMOut_PutS("GetBaseAddr(): sizeof u32 = "); VMOut_PutI((i32)sizeof(u32)); VMOut_PutN();
133
134     mem = GetBaseAddr(memAllocated, (u32)1024UL);
135 //t    VMOut_PutS("Admin: memAllocated = "); VMOut_PutX((u32)memAllocated); VMOut_PutN();
136 //t    VMOut_PutS("Admin: mem          = "); VMOut_PutX((u32)mem); VMOut_PutN();
137
138     /* Parse options */
139     for (; i < argc; i++) {
140         if ( (strcmp(argv[i], "-?") == 0) || (strcmp(argv[i], "-help") == 0) ) {
141             Usage();
142             return 0;
143         } else if (strcmp(argv[i], "-v") == 0) {
144             DisplayBanner();
145             return 0;
146         } else {
147             break;
148         }
149     }
150
151     /* Parse file */
152     if (i == argc-1) {
153         char *pfile;
154
155         strcpy(filename, argv[i]);   /* save name and extension */
156 //t        VMOut_PutS("Parse file: Filename: '%s'\n", filename);
157
158         name = GetFileName(filename);
159         ext  = GetFilenameExt(filename);
160         strcpy(filename, name);
161
162 //t        VMOut_PutS("Filename: '%s' Name: '%s' Ext: '%s':\n", filename, name, ext);
163
164         if (ext && (strcmp(ext, "exe") == 0)) {  /* 3 characters extension maximum */
165             char pb[50];
166
167             strcpy(pb, "");
168             pfile = strcat(pb, filename);
169
170 //t            VMOut_PutS("fopen: Filename: '%s'\n", pfile);
171
172             file = fopen(pfile, "rb" );
173             if (file == NULL) {
174                 VMOut_PutS(filename); VMOut_PutS(" does not exist.\n");
175                 return -1;
176             }
177
178             if (!loadObjFile(file, MemMax)) { // not a success because too big
179                 return -2;
180             }
181         } else {
182             VMOut_PutS("Error: Must have a file with '.exe' extension.\n");
183             Usage();
```

```
184             return -3;
185         }
186     } else {
187         VMOut_PutS("Error: Must have a file to load.\n");
188         Usage();
189         return -4;
190     }
191
192     VM_Init(mem);
193     VM_execute(mem);
194     return 0;
195 }
```

3.2 opcode.h

```
1   /* opcode.h
2   //
3   // Copyright (C) 1999-2020 by Michel de Champlain
4   //
5   */
6   #if !defined(OPCODE_H)
7   #    define  OPCODE_H
8
9   typedef enum {
10  // ---------------------------------------- INHERENT [0x00..0x2F]
11  // RFUn means Reserved for Future Used
12
13      HALT, POP,  DUP,  EXIT, RET, RFU1, RFU2, RFU3, // 0x00 .. 0x07
14      RFU4, RFU5, RFU6, RFU7, NOT,  AND,  OR,   XOR, // 0x08 .. 0x0F
15      NEG,  INC,  DEC,  ADD,  SUB,  MUL,  DIV,  REM, // 0x10 .. 0x17
16      SHL,  SHR,  TEQ,  TNE,  TLT,  TGT,  TLE,  TGE, // 0x18 .. 0x1F
17
18  INHERENT_END = 0x2F,
19
20  // ---------------------------------------- IMMEDIATE [0x30..0xAF]
21      BR_I5    = 0x30,
22      BRF_I5   = 0x50,
23      ENTER_U5 = 0x70,
24      LDC_I3   = 0x90,
25      ADDV_U3  = 0x98,
26       LDV_U3  = 0xA0,
27       STV_U3  = 0xA8,
28
29  // ---------------------------------------- RELATIVE [0xB0..0xFF]
30  RELATIVE_BEGIN = 0xB0,
31
32      ADDV_U8 = 0xB0,
33       LDV_U8 = 0xB1,
34       STV_U8 = 0xB2,
35      INCV_U8 = 0xB3,
36      DECV_U8 = 0xB4,
37
38     ENTER_U8 = 0xBF,
39
40       LDA_I8  = 0xD4,
41       LDA_I16 = 0xD5,
42
43       LDC_C8  = 0xD7,
44       LDC_C16 = 0xD8,
45       LDC_I8  = 0xD9,
46       LDC_I16 = 0xDA,
47       LDC_I32 = 0xDB,
48       LDC_U8  = 0xDC,
49       LDC_U16 = 0xDD,
50       LDC_U32 = 0xDE,
51
```

```
52          BR_I8  = 0xE0,
53          BR_I16 = 0xE1,
54         BRF_I8  = 0xE3,
55         BRF_I16 = 0xE4,
56
57       CALLS_I8  = 0xE6,
58       CALLS_I16 = 0xE7,
59
60        TRAP     = 0xFF,
61    RELATIVE_END = 0xFF,
62
63    } Opcode;
64
65    #endif /* OPCODE_H */
```

3.3 vm.h

```
1     /* vm.h - Cm Virtual Machine - Interface
2     //
3     // Copyright (C) 1999-2020 by Michel de Champlain
4     //
5     */
6
7     #ifndef __vm_h
8     #define __vm_h
9
10    #include "_stdtype.h"
11
12    void VM_Init(u8* mainAddr);
13    void VM_execute(u8* startAddr);
14
15    #endif
```

3.4 vm.c

```
1     /* vm.c - Cm Virtual Machine implementation
2     //
3     // Copyright (C) 1999-2020 by Michel de Champlain
4     //
5     */
6
7     #include "out.h"
8     #include "ioreg.h"
9     #include <string.h> /* strtok */
10    #include <stdlib.h> /* exit */
11
12    /* Cm EVM Error messages */
13    #define RESERVED_FOR_FUTURE_USED       " is reserved for future used."
14    #define FATAL_ERROR                    "Fatal error("
15    #define CANNOT_BE_ALLOCATED            "cannot be allocated.\n"
16
17    #include "opcode.h"
18    #include "vmstack.h"
19    #include "vm.h"
20
21    #ifdef IORegOn
22    #include "ioreg.h"
23    #endif
24
25    extern  u8*   mem; // Need the memory address for the loader.
26
27    #define SpInitial  200
28
29    static u32    dp;
30    static u32    vt;
```

```
31  static Stack  runningStack;
32
33  static void checkFlags() {
34  //t  if (flags & Flags_Stack) { Stack_print(runningStack); }
35  }
36
37  static StackEnumtor se;
38
39  void VM_Init(u8* mainAddr) {
40      u8 mainId;
41      dp = 0UL;
42
43  #define Thread_StackCapacity 32
44
45      mainId = 0;
46      runningStack = Stack_New(Thread_StackCapacity); // Only one thread in cm VM.
47      se = Stack_GetEnumerator(runningStack);
48
49  #ifdef MONITOR
50      Stack_print(runningStack); putln();
51  #endif
52  }
53
54  #ifdef InterruptManagerOn
55  //---------------------------------------------------------------------
56  // Interrupt Manager - first parameter is pushed first
57  //---------------------------------------------------------------------
58  static void InterruptManager(int op) {
59      u32 handlerAddr;
60      u8  number;
61
62      switch( op & 0x0F ) {
63          case 0: Interrupt_Disable(); break;
64          case 1: Interrupt_Enable(); break;
65          case 2: Stack_Push(runningStack, (u32)Interrupt_SaveAndDisable()); break;
66          case 3: Interrupt_Restore( Stack_Pop(runningStack) ); break;
67
68          case 4: /* void Interrupt_SetVector(u8 number, u32 handlerAddr) */
69              handlerAddr = (u32)Stack_Pop(runningStack);
70              number      =  (u8)Stack_Pop(runningStack);
71              Interrupt_SetVector(number, handlerAddr);
72              break;
73
74          case 5: /* u32  Interrupt_GetVector(u8 number) */
75              number = (u8)Stack_Pop(runningStack);
76              Stack_Push(runningStack, (u32)Interrupt_GetVector(number));
77              break;
78      }
79  }
80  #endif
81  //---------------------------------------------------------------------
82  // Kernel I/O Manager
83  //---------------------------------------------------------------------
84  static void Kernel_IO(int op) {
85  //t    VMOut_PutS("Kernel I/O: op = "); VMOut_PutX((i32)op & 0x0F); VMOut_PutN();
86  //t    VMOut_PutS("top = "); VMOut_PutX((int)Stack_top(runningStack)); VMOut_PutN();
87      switch( op & 0x0F ) {
88          case 0: /* Put(bool)    */ VMOut_PutB(      Stack_Pop(runningStack)); break;
89          case 1: /* Put(char)    */ VMOut_PutC((char)Stack_Pop(runningStack)); break;
90          case 2: /* Put(int)     */ VMOut_PutI(Stack_Pop(runningStack)); break;
91          case 3: /* Put(uint)    */ VMOut_PutU(Stack_Pop(runningStack)); break;
92  //      case 4: /*                  VMOut_PutF(): No support for float in Cm VM      */
93          case 5: /* Put(cstring) */ VMOut_PutS( (char*)Stack_Pop(runningStack) ); break;
94          case 6: /* PutHex(uint) */ VMOut_PutX(       Stack_Pop(runningStack));  break;
95          case 7: /* PutLine()    */ VMOut_PutN(); break;
96      }
97  }
98
```

```
99  static void exitOnInvalidOpcode(u8 opcode) {
100     VMOut_PutS("Opcode: "); VMOut_PutX(opcode); VMOut_PutS(RESERVED_FOR_FUTURE_USED); VMOut_PutN();
101     exit(0);
102 }
103 static void exitThreadNotSupported(void) {
104     VMOut_PutS("This version does not support threads.\n");
105     exit(0);
106 }
107
108 void VM_execute(u8* startAddr) {
109     u8  opcode;
110     u8* ip;
111
112 //t VMOut_PutS("sizeof u8* ip = "); VMOut_PutU((u32)sizeof(ip)); VMOut_PutN();
113
114     for (ip = startAddr; opcode = *ip;) {
115 //t        VMOut_PutS("ip = "); VMOut_PutX((u32)ip); VMOut_PutS("*ip = "); VMOut_PutX((u32)opcode); VMOut_PutN();
116
117         if ( opcode > HALT  &&  opcode <= INHERENT_END ) {
118             switch(opcode) {
119                 case POP:    Stack_Pop(runningStack); ip++; break;
120                 case DUP:    Stack_dup(runningStack); ip++; break;
121                 case EXIT:   Stack_exit(runningStack); ip = (u8*)Stack_Pop(runningStack); break;
122                 case RET:    ip = (u8*)Stack_Pop(runningStack); break;
123                 case NOT:    Stack_not(runningStack);   ip++; break;
124                 case AND:    Stack_and(runningStack);   ip++; break;
125                 case OR:     Stack_or (runningStack);   ip++; break;
126                 case XOR:    Stack_xor(runningStack);   ip++; break;
127                 case NEG:    Stack_neg(runningStack);   ip++; break;
128                 case INC:    Stack_inc(runningStack);   ip++; break;
129                 case DEC:    Stack_dec(runningStack);   ip++; break;
130                 case ADD:    Stack_add(runningStack);   ip++; break;
131                 case SUB:    Stack_sub(runningStack);   ip++; break;
132                 case MUL:    Stack_mul(runningStack);   ip++; break;
133                 case DIV:    Stack_div(runningStack);   ip++; break;
134                 case REM:    Stack_rem(runningStack);   ip++; break;
135                 case SHL:    Stack_shl(runningStack);   ip++; break;
136                 case SHR:    Stack_shr(runningStack);   ip++; break;
137                 case TEQ:    Stack_teq(runningStack);   ip++; break;
138                 case TNE:    Stack_tne(runningStack);   ip++; break;
139                 case TLT:    Stack_tlt(runningStack);   ip++; break;
140                 case TGT:    Stack_tgt(runningStack);   ip++; break;
141                 case TLE:    Stack_tle(runningStack);   ip++; break;
142                 case TGE:    Stack_tge(runningStack);   ip++; break;
143             }
144         } else if ( opcode >= BR_I5     &&  opcode < BRF_I5   ) {
145             i8 offset = opcode - BR_I5;
146             if (offset >= 0x10) offset = (offset - 0x10) | 0xFFFFFFF0;
147             ip += offset;
148 //t            VMOut_PutS("xBR_I5 at "); VMOut_PutX((u32)ip); VMOut_PutN();
149         } else if ( opcode >= BRF_I5    &&  opcode < ENTER_U5 ) {
150             i8 offset = opcode - BRF_I5;
151             if (offset >= 0x10) offset = (offset - 0x10) | 0xFFFFFFF0;
152             if (Stack_Pop(runningStack)) offset = 1; /* no branching if true */
153             ip += offset;
154 //t            VMOut_PutS("xBRF_I5 at "); VMOut_PutX((u32)ip); VMOut_PutN();
155         } else if ( opcode >= ENTER_U5  &&  opcode < LDC_I3   ) {
156             u8 fctInfo = opcode - ENTER_U5;
157             Stack_enterU5(runningStack, fctInfo);
158             ip++;
159         } else if ( opcode >= LDC_I3    &&  opcode < ADDV_U3  ) {
160             short constant = (opcode & 0x07) % 0x08;
161             if (constant >= 0x04) constant = (constant - 0x04) | 0xFFFC;
162             Stack_Push(runningStack, (i16)constant);
163 #ifdef ExampleOfStackDump
164     se = Stack_GetEnumerator(runningStack);
165     Stack_Print(se);
166     VMOut_PutS("\n");
```

```
167 #endif
168             ip++;
169         } else if ( opcode >= ADDV_U3   &&  opcode < LDV_U3   ) {
170             Stack_addVariable(runningStack, opcode & 0x07);
171             ip++;
172         } else if ( opcode >= LDV_U3    &&  opcode < STV_U3   ) {
173             Stack_loadVariable(runningStack, opcode & 0x07);
174             ip++;
175         } else if ( opcode >= STV_U3    &&  opcode < ADDV_U8  ) {
176             Stack_storeVariable(runningStack, opcode & 0x07);
177             ip++;
178         } else if (opcode >= RELATIVE_BEGIN && opcode <= RELATIVE_END) {
179             switch(opcode) {
180             case ADDV_U8: {
181                 Stack_addVariable(runningStack, *(ip + 1));
182                 ip += 2;
183                 break;
184             }
185             case LDV_U8: {
186                 Stack_loadVariable(runningStack, *(ip + 1));
187                 ip += 2;
188                 break;
189             }
190             case STV_U8: {
191                 Stack_storeVariable(runningStack, *(ip + 1));
192                 ip += 2;
193                 break;
194             }
195             case INCV_U8: {
196                 Stack_Push(runningStack, 1);
197                 Stack_addVariable(runningStack, *(ip + 1));
198                 ip += 2;
199                 break;
200             }
201             case DECV_U8: {
202                 Stack_Push(runningStack, -1);
203                 Stack_addVariable(runningStack, *(ip + 1));
204                 ip += 2;
205                 break;
206             }
207             case ENTER_U8: {
208                 Stack_enterU8(runningStack, *(ip + 1));
209                 ip += 2;
210                 break;
211             }
212             case LDA_I8: {
213                 i8 offset = *(ip+1);
214                 Stack_Push(runningStack, (i32)(ip+offset));
215                 ip += 2;
216                 break;
217             }
218             case LDA_I16: {
219                 i16 offset = (*(ip+1) << 8) | *(ip+2);
220                 Stack_Push(runningStack, (i32)(ip+offset));
221                 ip += 3;
222                 break;
223             }
224             case LDC_C8:
225             case LDC_I8:
226             case LDC_U8: {
227                 u8   u8const = 0;
228                 i8   i8const = 0;
229 
230                 if (opcode == LDC_I8) i8const = *(ip+1); else u8const = *(ip+1);
231                 Stack_Push(runningStack, (opcode == LDC_I8) ? (i32)i8const : (u32)u8const);
232                 ip += 2;
233                 break;
234             }
```

```
235             case LDC_C16:
236             case LDC_I16:
237             case LDC_U16: {
238                 u16  u16const = 0;
239                 i16  i16const = 0;
240 
241                 if (opcode == LDC_I16) {
242                     i16const = (*(ip + 1) << 8) | *(ip + 2);
243                 } else {
244                     u16const = (*(ip + 1) << 8) | *(ip + 2);
245                 }
246                 Stack_Push(runningStack, (opcode == LDC_I16) ? (i32)i16const : (u32)u16const);
247                 ip += 3;
248                 break;
249             }
250             case  LDC_I32:
251             case  LDC_U32: {
252                 u8  msb = *++ip;
253                 u8  lsb = *++ip;
254                 u32 lsoffset, msoffset = (u32)( ((msb << 8) & 0x0000FF00L) | (lsb & 0x000000FFL));
255                 i32 offset;
256 
257                 msb = *++ip;
258                 lsb = *++ip;
259                 lsoffset = (u32)( ((msb << 8) & 0x0000FF00L) | (lsb & 0x000000FFL));
260 
261                 offset = (i32)( (msoffset << 16) | lsoffset );
262 
263                 Stack_Push(runningStack, offset);
264                 ip++;
265                 break;
266             }
267             case BR_I8: {
268                 i8 relAddr = *(ip+1);
269                 ip += relAddr;
270                 break;
271             }
272             case BRF_I8: {
273                 i8 relAddr;
274                 if (Stack_Pop(runningStack))
275                     relAddr = 2; /* no branching if true */
276                 else
277                     relAddr = *(ip+1); /* false, then branch */
278 
279                 ip += relAddr;
280                 break;
281             }
282             case BR_I16: {
283                 i16 relAddr = (*(ip+1) << 8) | *(ip+2);
284                 ip += relAddr;
285                 break;
286             }
287             case CALLS_I16: {
288                 i16 relAddr = (*(ip+1) << 8) | *(ip+2);
289                 Stack_Push(runningStack, (i32)(ip+3));
290                 ip += relAddr;
291 //t                VMOut_PutS("calls_16 (ip = "); VMOut_PutX((i32)ip); VMOut_PutN();
292                 break;
293             }
294             case TRAP: {
295                 u8 op = *(ip+1);
296                 switch( op ) {
297 #ifdef InterruptManagerOn
298                     // Interrupt Manager
299                     case 0x00: case 0x01: case 0x02: case 0x03: case 0x04: case 0x05:
300                         InterruptManager(op);
301                         break;
302 #endif
```

```
303                     /* Kernel I/O */
304                     case 0x80: case 0x81: case 0x82: case 0x83:
305                     case 0x84: case 0x85: case 0x86: case 0x87:
306                         Kernel_IO(op);
307                         break;
308                 }
309                 ip += 2;
310                 break;
311             }
312             default:
313                 exitOnInvalidOpcode(opcode);
314             }
315         }
316 //---------------------------------
317 //t        Stack_Print(se);
318         checkFlags();
319     }
320 }
```