

```
1: //-----
2: #ifndef UI1410INSTH
3: #define UI1410INSTH
4: //-----
5:
6: #include "ubcd.h"
7: #include "UI1410CPUH.H"
8: #include "UI1410DEBUG.H"
9:
10: #include <assert.h>
11:
12: // Op code table Instruction Readout lines values: OpReadoutLines
13:
14: #define OP_PERCENTTYPE 1
15: #define OP_NOTPERCENTTYPE 2
16: #define OP_ADDRDBL 4
17: #define OP_NOTADDRDBL 8
18: #define OP_1ADDRPLUSMOD 16
19: #define OP_2ADDRNOMOD 32
20: #define OP_2ADDRPLUSMOD 64
21: #define OP_2ADDRESS 128
22: #define OP_ADDRTYPE 256
23: #define OP_2CHARONLY 512
24: #define OP_CCYCLE 1024
25: #define OP_NOCORDCY 2048
26: #define OP_NODCYIRING6 4096
27: #define OP_NOINDEXON1STADDR 8192
28:
29: // Op code table Operational lines values: OpOperationalLines
30:
31: #define OP_RESETTYPE 1
32: #define OP_ADDORSUBT 2
33: #define OP_MPYORDIV 4
34: #define OP_ADDTYPE 8
35: #define OP_ARITHTYPE 16
36: #define OP_EORZ 32
37: #define OP_COMPARETYPE 64
38: #define OP_BRANCHTYPE 128
39: #define OP_NOBRANCH 256
40: #define OP_WORDMARK 512
41: #define OP_MORL 1024
42:
43: // Op code table Control lines values: OpControlLines
44:
45: #define OP_1STSCANFIRST 1
46: #define OP_ACYFIRST 2
47: #define OP_STDACYCLE 4
48: #define OP_BCYFIRST 8
49: #define OP_AREGTOACHONBCY 16
50: #define OP_OPMODTOACHONBCY 32
51: #define OP_LOADMEMONBCY 64
52: #define OP_RGENMEMONBCY 128
53: #define OP_STOPATFONBCY 256
54: #define OP_STOPATHONBCY 512
55: #define OP_STOPATJONBCY 1024
56: #define OP_ROBARONSCANBCY 2048
57: #define OP_ROAARONACY 4096
58:
59: #define OP_INVALID 65535
60:
61: #define OP_SUBTRACT 18
62: #define OP_TABLESEARCH 19
63: #define OP_NOP 37
64: #define OP_ZERO_SUB 42
65: #define OP_ADD 49
66: #define OP_SAR_G 55
```

```
67: #define OP_ZERO_ADD      58
68: #define OP_MULTIPLY     12
69: #define OP_DIVIDE        28
70: #define OP_MOVE          52
71: #define OP_MCS           25
72: #define OP_EDIT          53
73:
74:
75: #endif
```

variables  
OP-NUO-SYMBOC

OD

```

1: //-----
2: #include <vcl\vcl.h>
3: #pragma hdrstop
4:
5: #include "UI1410INST.H"
6:
7: //-----
8:
9: #include "UI1415IO.H"
10:
11: // This module handles Instruction Decode and Execution in the CPU
12:
13: /* The following table is given in the order of the 1410 BCD codes, and
14: contains the opcode common lines - 3 16 bit words.
15: */
16:
17: struct OpCodeCommonLines OpCodeTable[64] = {
18:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 00 spc */
19:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 01 1 */
20:     { 8+512, 256, 32+128 }, /* 02 2 */
21:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 03 3 */
22:     { 8+512, 256, 32+128 }, /* 04 4 */
23:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 05 5 */
24:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 06 6 */
25:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 07 7 */
26:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 08 8 */
27:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 09 9 */
28:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 10 0 */
29:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 11 = */
30:     { 2+8+32+128+256+1024, 4+16+256, 1+2+4+16+64+256+1024 }, /* 12 @ */
31:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 13 : */
32:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 14 > */
33:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 15 rad */
34:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 15 alt */
35:     { 2+4+32+128+256+2048+4096, 128, 1+8+16+256+2048 }, /* 17 J */
36:     { 2+4+32+128+256+4096, 2+8+16+256, 1+2+4+16+64+4096 }, /* 18 S */
37:     { 2+8+64+128+256+1024+4096, 64+256, 1+2+4+16+128+2048 }, /* 19 T */
38:     { 1+8+16+256+8192, 256, 32 }, /* 20 U */
39:     { 2+8+64+128+256+2048+4096, 128, 1+8+32+128+512+2048 }, /* 21 V */
40:     { 2+8+64+128+256+2048+4096, 128, 1+8+32+128+512+2048 }, /* 22 W */
41:     { 2+4+16+256, 128, 32+128+256+2048 }, /* 23 X */
42:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 24 Y */
43:     { 2+8+32+128+256+2048+4096, 32+256, 1+2+4+16+64+1024+2048+4096 }, /* 25 Z */
44:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 26 RM */
45:     { 2+4+32+128+256+2048+4096, 256+512, 1+2+16+2048+4096 }, /* 27 , */
46:     { 2+8+32+128+256+1024, 4+16+256, 1+2+4+16+64+256+1024 }, /* 28 % */
47:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 29 WS */
48:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 30 \ */
49:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 31 SM */
50:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 32 - */
51:     { 2+4+16+256+2048, 128, 32+128+256+2048 }, /* 33 J */
52:     { 8+512, 256, 32+128 }, /* 34 K */
53:     { 1+8+64+128+256+8192, 256+1024, 0 }, /* 35 L */
54:     { 1+8+64+128+256+8192, 256+1024, 0 }, /* 36 M */
55:     { 0, 0, 0 }, /* 37 N */
56:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 38 O */
57:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 39 P */
58:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 40 Q */
59:     { 2+4+16+256, 128, 32+128+256+2048 }, /* 41 R */
60:     { 2+4+32+128+256+4096, 1+8+16+256, 1+2+4+16+64+256+1024+4096 }, /* 42 ! */
61:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 43 $ */
62:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 44 * */
63:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 45 ] */
64:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 46 ; */
65:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 47 Delt */
66:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 48 + */

```

```
67: { 2+4+32+128+256+4096, 2+8+16+256, 1+2+4+16+64+4096 }, /* 49 A */
68: { 2+8+64+128+256+2048+4096, 64+128, 1+8+32+128+512+2048 }, /* 52 B */
69: { 2+8+32+128+256+2048+4096, 64+256, 1+2+4+16+128+2048+4096 }, /* 51 C */
70: { 2+8+64+128+256+2048+4096, 256, 2+4+16+64+256+1024+2048+4096 }, /* 52 D */
71: { 2+8+32+128+256+2048+4096, 32+256, 1+2+4+16+64+1024+2048+4096 }, /* 53 E */
72: { 8+512, 256, 32+128 }, /* 54 F */
73: { 8+16+256+8192, 256, 1 }, /* 55 G */
74: { OP_INVALID, OP_INVALID, OP_INVALID }, /* 56 H */
75: { OP_INVALID, OP_INVALID, OP_INVALID }, /* 57 I */
76: { 2+4+32+128+256+4096, 1+8+16+256, 1+2+4+16+64+256+1024+4096 }, /* 58 ? */
77: { 2+8+256+2048+4096, 128, 16+128+256+2048 }, /* 59 . */
78: { 2+4+32+128+256+2048+4096, 256+512, 1+2+16+2048+4096 }, /* 60 loz. */
79: { OP_INVALID, OP_INVALID, OP_INVALID }, /* 61 [ */
80: { OP_INVALID, OP_INVALID, OP_INVALID }, /* 62 < */
81: { OP_INVALID, OP_INVALID, OP_INVALID } /* 63 GM */

82: };
83:
84: // Table indicating when zones are valid for ops with addresses
85:
86: static bool IRingZoneTable [] = {
87:     false, false, false, true, true, false, false, true, true,
88:     false, true, true
89: };
90:
91:
92: // Table of Index Register locations
93:
94: int IndexRegisterLookup[16] = {
95:     0,29,34,39,44,49,54,59,64,69,74,79,84,89,94,99
96: };
97:
98: // START BUTTON pressed - moved from UI1410PWR to here
99:
100: void T1410CPU::DoStartClick()
101: {
102:     switch (Mode) {
103:
104:         case MODE_ADDR:
105:             ProcessRoutineLatch = false;
106:             FI1415IO -> DoAddressEntry();
107:             BranchTo1Latch = false;
108:             BranchLatch = false;
109:             break;
110:
111:         case MODE_DISPLAY:
112:             ProcessRoutineLatch = false;
113:             FI1415IO -> DoDisplay(1);
114:             break;
115:
116:         case MODE_ALTER:
117:             ProcessRoutineLatch = false;
118:             FI1415IO -> DoAlter();
119:             break;
120:
121:         case MODE_IE:
122:         case MODE_RUN:
123:
124:             // We loop here until something makes us stop and return,
125:             // such as a special cycle control setting, I/E mode when
126:             // finished fetching, the STOP key at the end of an instruction,
127:             // etc.
128:
129:             StopLatch = false;
130:
131:             // These are debugging statements. I've had a lot of problems
132:             // using ptr = ptr instead of *ptr = *ptr
```

```

133:         assert(STAR != A_AR);
134:         assert(STAR != B_AR);
135:         assert(STAR != C_AR);
136:         assert(STAR != D_AR);
137:         assert(A_AR != B_AR);
138:         assert(A_AR != C_AR);
139:         assert(A_AR != D_AR);
140:         assert(B_AR != C_AR);
141:         assert(B_AR != D_AR);
142:         assert(C_AR != D_AR);
143:
144:     while(true) {
145:
146:         // I Cycle start
147:
148:         ProcessRoutineLatch = true;
149:
150:         if(IRingControl) {
151:             InstructionDecodeStart();
152:             if(StopLatch) {
153:                 FI1415IO -> StopPrintOut('S');
154:                 return;
155:             }
156:             if(CycleControl != CYCLE_OFF ||
157:                 (LastInstructionReadout && Mode == MODE_IE)) {
158:                 FI1415IO -> StopPrintOut('C');
159:                 break;
160:             }
161:         }
162:
163:         // I Cycle
164:
165:         else if(!LastInstructionReadout &&
166:                 CycleRing -> State() == CYCLE_I) {
167:             InstructionDecode();
168:             if(StopLatch) {
169:                 FI1415IO -> StopPrintOut('S');
170:                 return;
171:             }
172:             if(CycleControl != CYCLE_OFF ||
173:                 (LastInstructionReadout && Mode == MODE_IE)) {
174:                 FI1415IO -> StopPrintOut('C');
175:                 break;
176:             }
177:         }
178:
179:         // X (index) Cycle
180:
181:         else if(!LastInstructionReadout &&
182:                 CycleRing -> State() == CYCLE_X) {
183:             InstructionIndex();
184:             if(StopLatch) {
185:                 FI1415IO -> StopPrintOut('S');
186:                 return;
187:             }
188:             if(CycleControl != CYCLE_OFF) {
189:                 FI1415IO -> StopPrintOut('C');
190:                 break;
191:             }
192:         }
193:
194:         // Execute Cycle would go here....
195:
196:
197:         if(LastInstructionReadout ||
198:             (!IRingControl && CycleRing -> State() != CYCLE_I) ) {

```

// StopLatch

if (StopKeyLatch) {  
 StopLatch = true;  
 StopKeyLatch = false;  
 FI1415IO -> StopPrintOut('S');  
 return;

B L      N      N      X

```
199: // Temporarily use LastInstructionReadout to set IRingControl
200: /*          // In otherwords, all instructions do nothing.
201:
202:
203:          CPU -> IRingControl = true;
204:          CPU -> LastInstructionReadout = false;
205: */
206:
207:          // Try and execute the instruction. Note that we typically
208:          // do this in the *same* cycle as the one where instruction
209:          // readout completes.
210:
211:          // LastInstructionReadout is still set the first time thru.
212:          // The execute routine must set the CPU to some cycle other
213:          // than I!
214:
215:          InstructionExecuteRoutine[Op_Reg -> Get().ToInt() & 0x3f]();
216:
217:          LastInstructionReadout = false;
218:
219:          if(StopLatch) {
220:              FI1415IO -> StopPrintOut('S');
221:              return;
222:          }
223:
224:          // If we are in storage/logic cycle, break out.
225:
226:          if(CycleControl != CYCLE_OFF) {
227:              FI1415IO -> StopPrintOut('C');
228:              break;
229:          }
230:
231:          // If we aren't in Storage Cycle or I/E Cycle Mode...
232:
233:          // Give Windoze a chance to breathe
234:
235:          Application -> ProcessMessages();
236:
237:      }
238:
239:      break;
240:
241:      default:
242:          break;
243:      }
244:
245:  }
246:
247:
248:
249: // Instruction Decode - initial phase
250:
251: void T1410CPU::InstructionDecodeStart()
252: {
253:
254:     int op_bin;
255:
256:     SetScan(SCAN_N);           // During I phase, no storage scan mode set
257:     CycleRing -> Set(CYCLE_I); // Doing I cycles
258:     IRing -> Reset();         // Reset I Ring to Op state
259:     LastInstructionReadout = false; // Set true at end of fetch
260:     IRingControl = false;       // Reset I Ring control state
261:
262:     // Figure out where to go. If we are branching, then branch to AAR
263:     // unless the BranchToILatch (e.g. Program Reset) is set. If not
264:     // branching, just use the IAR. STAR is set to the location to begin
```

```
265:     // instruction readout.
266:
267:     if(BranchLatch) {
268:         if(BranchToILatch) {
269:             STAR -> Set(1);
270:         }
271:         else {
272:             *STAR = *A_AR;
273:         }
274:     }
275:     else {
276:         *STAR = *I_AR;
277:     }
278:
279:     // Fetch the instruction code. Check to make sure it has it's wordmark
280:     // If not, stop with an instruction check.
281:
282:     // Take an I Cycle
283:
284:     Readout();
285:     Cycle();
286:     I_AR -> Set(STARMod(+1));           // This one *always* advances I_AR
287:
288:     if(!B_Reg -> Get().TestWM()) {
289:         InstructionCheck -> SetStop("Instruction Check: No WordMark present");
290:         return;
291:     }
292:
293:     // Copy the opcode into the Op register, and decode it.
294:
295:     *Op_Reg = *B_Reg;
296:
297:     op_bin = Op_Reg -> Get().ToInt() & 0x3f;
298:     OpReadOutLines = OpCodeTable[op_bin].ReadOut;
299:     OpOperationalLines = OpCodeTable[op_bin].Operational;
300:     OpControlLines = OpCodeTable[op_bin].Control;
301: }
302:
303: // Instruction Decode
304:
305: void T1410CPU::InstructionDecode()
306: {
307:     int op_bin;
308:     BCD b;
309:
310:     // The initial state of the ring gets special handling for NOP
311:
312:     if(IRing -> State() == I_RING_OP) {
313:
314:         BranchToILatch = BranchLatch = false;
315:
316:         // Take an I Cycle
317:
318:         *STAR = *I_AR;
319:         Readout();                      // Next inst. character now in B Register
320:         Cycle();
321:
322:         // In the real machine, what happens is that if a character past the
323:         // op code is read out contained a WM, I_AR is not set, but is then
324:         // later set from STAR. We handle that this way here.
325:
326:         if(!B_Reg -> Get().TestWM()) {
327:             I_AR -> Set(STARMod(+1));
328:         }
329:
330:         // If this is a NOP, it gets special handling right here.
```

```

331:     // If there is no wordmark, we do nothing, and come right back
332:     // here with the next cycle (to check for a WM again).
333:     // If there is a WordMark, we decode the op.
334:
335:     if((Op_Reg -> Get() & 0x3f) == OP_NOP) {
336:
337:         if(B_Reg -> Get().TestWM()) {
338:             // Copy the opcode into the Op register, and decode it.
339:             *Op_Reg = *B_Reg;
340:
341:             op_bin = Op_Reg -> Get().ToInt() & 0x3f;
342:             OpReadOutLines = OpCodeTable[op_bin].ReadOut;
343:             OpOperationalLines = OpCodeTable[op_bin].Operational;
344:             OpControlLines = OpCodeTable[op_bin].Control;
345:
346:         }
347:         return;           IAR → S+(STAR_M1 (+1));
348:     }
349:
350:
351:     // If there are no control lines set, it is an invalid op code
352:
353:     if(OpReadOutLines == OP_INVALID || OpOperationalLines == OP_INVALID ||
354:        OpControlLines == OP_INVALID) {
355:         InstructionCheck -> SetStop("Instruction Check: Invalid OP code");
356:         return;
357:     }
358:
359:     // else Not a NOP. Advance I Ring
360:
361:     IRing -> Next();
362:
363:     // For % and lozenge ops (that use channel ID), set I Ring to 3
364:
365:     if(OpReadOutLines & OP_PERCENTTYPE) {
366:         IRing -> Set(I_RING_3);
367:     }
368:
369:     // Proceed to "D" on the next cycle
370:
371:     return;
372: }
373:
374: // At IRing 1 and IRing 6, the index latches are reset.
375:
376: if(IRing -> State() == 1 || IRing -> State() == 6) {
377:     IndexLatches = 0;
378: }
379: // Entry point "D"          ← IRig 8 or M1 is Reset 2nd Pg 26
380: // If we have a wordmark, handle Instruction length checking...
381:
382: if(B_Reg -> Get().TestWM()) {
383:
384:     if(B_Reg -> Get().TestWM()) {
385:
386:         // Now, since the B_Reg has a WM, we need to restore I_AR from
387:         // STAR, just like in the real machine, so that IAR points to the
388:         // following opcode. (Otherwise, it would point one past the opcode)
389:
390:         *I_AR = *STAR;
391:
392:         switch(IRing -> State()) {
393:
394:             case I_RING_1:
395:
396:                 // Handle opcode with no address or which don't need

```

PG. Pg 12

```
397:         // CAR or DAR to chain.
398:
399:         if(OpReadOutLines & OP_NOCORDCY) {
400:             LastInstructionReadout = true;
401:             return;
402:         }
403:
404:         // Handle chaining of arithmetic type op codes
405:
406:         if(OpOperationalLines & OP_ARITHTYPE) {
407:             CycleRing -> Set(CYCLE_D);
408:             *STAR = *B_AR;
409:             *D_AR = *STAR;      // Mod by 0
410:
411:             // If Multiply or Divide, also set CAR
412:
413:             if(OpOperationalLines & OP_MPYORDIV) {
414:                 CycleRing -> Set(CYCLE_D);
415:                 *STAR = *A_AR;
416:                 *C_AR = *STAR;    // Mod by 0
417:             }
418:
419:             LastInstructionReadout = true;
420:             return;
421:         }
422:
423:         // Handle chaining of Table Lookup
424:
425:         if((Op_Reg -> Get() & 0x3f) == OP_TABLESEARCH) {
426:             CycleRing -> Set(CYCLE_C);
427:             *STAR = *A_AR;
428:             *C_AR = *STAR;      // Mod by 0
429:             LastInstructionReadout = true;
430:             return;
431:         }
432:
433:         // Otherwise, an invalid 1 character opcode
434:
435:         InstructionCheck -> SetStop("Instruction Check: Invalid length at I1");
436:         return;
437:
438:     case I_RING_2:
439:
440:         // Handle simple 2 character op codes
441:
442:         if(OpReadOutLines & OP_2CHARONLY) {
443:             LastInstructionReadout = true;
444:             return;
445:         }
446:
447:         // Otherwise, we have something invalid
448:
449:         InstructionCheck -> SetStop("Instruction Check: Invalid length at I2");
450:         return;
451:
452:     case I_RING_6:
453:
454:         // First, handle ops that end or chain normally with 1 address
455:
456:         if(OpReadOutLines & OP_NODCYIRING6) { ~ if((Op_Reg -> Get() & 0x3f) == OP_HALT) {
457:             LastInstructionReadout = true;
458:             return;
459:         }
460:
461:         // Handle Multiply/Divide chaining
462: }
```

```
463:         if(OpOperationalLines & OP_MPYORDIV) {
464:             CycleRing -> Set(CYCLE_D);
465:             *STAR = *B_AR;
466:             *C_AR = *STAR; // Mod by 0
467:             LastInstructionReadout = true;
468:             return;
469:         }
470:         // Otherwise, something is wrong.
471:
472:         InstructionCheck -> SetStop("Instruction Check: Invalid length at I6");
473:         return;
474:
475:     case I_RING_7:
476:
477:         // Handle opcodes that are 1 character plus Op Modifier
478:
479:         if(OpReadOutLines & OP_1ADDRPLUSMOD) {
480:             LastInstructionReadout = true;
481:             return;
482:         }
483:
484:         // Otherwise, something is wrong
485:
486:         InstructionCheck -> SetStop("Instruction Check: Invalid Length at I7");
487:         return;
488:
489:     case I_RING_11:
490:
491:         // Handle op codes that have 2 addresses with no Op Modifier
492:
493:         if(OpReadOutLines & OP_2ADDRNOMOD) {
494:             LastInstructionReadout = true;
495:             return;
496:         }
497:
498:         // Otherwise, something is wrong.
499:
500:         InstructionCheck -> SetStop("Instruction Check: Invalid Length at I11");
501:         return;
502:
503:
504:     case I_RING_12:
505:
506:         // Handle op codes that have 2 addresses plus an Op Modifier
507:
508:         if(OpReadOutLines & OP_2ADDRPLUSMOD) {
509:             LastInstructionReadout = true;
510:             return;
511:         }
512:
513:         InstructionCheck -> SetStop("Instruction Check: Invalid Length at I12");
514:         return;
515:
516:     default:
517:
518:         InstructionCheck -> SetStop("Instruction Check: Invalid Length");
519:         return;
520:     }
521:
522: }
523:
524: // Make sure we don't fall thru here by accident.
525:
526: assert(!LastInstructionReadout && !(B_Reg -> Get().TestWM()));
527:
528: // End of handling of B Channel WordMark
```

```
529:
530:    // If this opcode should have addresses, check to make sure that
531:    // the instruction isn't too long...
532:
533:    if(OpReadOutLines & OP_ADDRTYPE) {
534:        if(OpReadOutLines & OP_2ADDRESS) {
535:            if(OpReadOutLines & OP_2ADDRNOMOD) {
536:                if(IRing -> State() == I_RING_11) {
537:                    InstructionCheck ->
538:                        SetStop("Instruction Check: 2 Addr too long at I11");
539:                    return;
540:                }
541:                // OK, proceed to "E"
542:            }
543:            else {
544:                assert(OpReadOutLines & OP_2ADDRPLUSMOD);
545:                if(IRing -> State() == I_RING_12) {
546:                    InstructionCheck ->
547:                        SetStop("Instruction Check: 2 Addr + Mod too long at I12");
548:                    return;
549:                }
550:                // OK, Proceed to "E"
551:            }
552:        }
553:        else if(OpReadOutLines & OP_1ADDRPLUSMOD) {
554:            if(IRing -> State() == 7) {
555:                InstructionCheck ->
556:                    SetStop("Instruction Check: 1 Addr + Mod too long at I7");
557:                return;
558:            }
559:            // OK, Proceed to "E"
560:        }
561:        else if(IRing -> State() == 6) {
562:            InstructionCheck ->
563:                SetStop("Instruction Check: 1 Addr too long at I6");
564:            return;
565:        }
566:        // OK, Proceed to "E"
567:    }
568:
569:    else if(OpReadOutLines & OP_2CHARONLY) {
570:
571:        if(IRing -> State() == 14) {
572:            *Op_Mod_Reg = *B_Reg;
573:            InstructionDecodeIARAdvance();
574:            return;
575:        }
576:
577:        else {
578:            InstructionCheck ->
579:                SetStop("Instruction Check: 2 Char Only too long at I2");
580:            return;
581:        }
582:    }
583:
584:    // ALL ops are either address type or 2 character!
585:
586:    else {
587:        InstructionCheck -> SetStop("Instruction Check: Not Addr or 2 Char ???");
588:        return;
589:    }
590:
591:    // Entry point "E" - no wordmark
592:
593:    // First, handle the I/O stuff (Percent type ops)
594:
```

```

595:     if(OpReadOutLines & OP_PERCENTTYPE) {
596:
597:         // The sections of code doing address validity checking show up
598:         // under entry point "C" on the flowchart, but since this
599:         // section doesn't end up there, we need to do it here.
600:
601:
602:     switch(IRing -> State()) {
603:
604:     case I_RING_3:           Channel[IOChannelSelect] → ChInterlock → Step C
605:
606:         if(IOMoveModeLatch || IOLeadModeLatch) {
607:             IOInterlockCheck ->
608:                 SetStop("I/O Interlock Check: I/O in progress at I3");
609:             return;
610:
611:
612:         // IO channel/overlap indicator must have 84 bit configuration
613:
614:         if((B_Reg -> Get() & BIT_NUM) != 0x0c) {
615:             AddressChannelCheck ->
616:                 SetStop("Address Channel Check: I/O Ch/Ovlp must have 84
config");
617:             return;
618:
619:         }
620:         IOChannelSelect = ((B_Reg -> Get() & BITB) != 0);
621:         InstructionDecodeIARAdvance();
622:         return;
623:
624:     case I_RING_4:
625:
626:         *(Channel[IOChannelSelect] -> ChUnitType) = *B_Reg;
627:         InstructionDecodeIARAdvance();
628:         return;
629:
630:     case I_RING_5:
631:
632:         // Unit number is not allowed to have zones
633:
634:         if(B_Reg -> Get().ToInt() & BIT_ZONE) {
635:             AddressZoneCheck ->
636:                 SetStop("Address Zone Check: Zones over unit number");
637:         }
638:
639:         *(Channel[IOChannelSelect] -> ChUnitNumber) = *B_Reg;
640:         InstructionDecodeIARAdvance();
641:         return;
642:
643:     default:
644:
645:         assert(IRing -> State() > 5);
646:
647:         break; // Continue on, knowing that for an I/O op we are
648:               // in I6 and above, so we will go to step "1" soon.
649:
650:     } // End IRing switch for Percent Type ops
651:
652: } // End Percent Type ops
653:
654: // Check to see if we are handling the 2nd address for 2 address
655: // ops (or the only address for I/O ops) (Step 1 on page 45)
656:
657: assert(IRing -> State() > 0);
658:
659: if(IRing -> State() > 5) {

```

*✓* *IOChannelSelect* → *ChInterlock* → *Step C*

*✓* *Check <= MAX!*  
*Just Check.*

*✓* *IOChannelSelect* → *ChUnitType*;

```
660:         if(OpReadOutLines & OP_1ADDRPLUSMOD) {
661:             *Op_Mod_Reg = *B_Reg;
662:             if(OpOperationalLines & OP_BRANCHTYPE) {
663:                 // Branch Handling will go here!
664:                 InstructionDecodeIARAdvance();
665:                 return;
666:             }
667:         }
668:     else {
669:         InstructionDecodeIARAdvance();
670:         return;
671:     }
672: }
673:
674: assert(OpReadOutLines & OP_2ADDRESS);
675:
676: // For 2 address ops, and for I/O operations,
677: // snag op modifier at I11.
678: // (Invalid lengths are checked before we get here).
679:
680: if(IRing -> State() > I_RING_10) {
681:     *Op_Mod_Reg = *B_Reg;
682:     if(OpOperationalLines & OP_BRANCHTYPE) {
683:         // Branch Handling will go here!
684:         InstructionDecodeIARAdvance();
685:         return;
686:     }
687:     else {
688:         InstructionDecodeIARAdvance();
689:         return;
690:     }
691: }
692:
693: // If we are at I6, we are starting address: reset BAR, DAR
694:
695: if(IRing -> State() == I_RING_6) {
696:     B_AR -> Reset();
697:     D_AR -> Reset();
698: }
699:
700: // Set the appropriate address character into B & DAR
701:
702: b = B_Reg -> Get();
703: b = b & BIT_NUM;
704: b.SetOddParity();
705:
706: B_AR -> Set(TWOOF5(b),IRing -> State() - 5);
707: D_AR -> Set(TWOOF5(b),IRing -> State() - 5);
708:
709: if(IRing -> State() == I_RING_10 &&
710:     IndexLatches > 0) {
711:     InstructionIndexStart();           // Start up indexing
712:     return;
713: }
714:
715: // Fall thru to "C"
716:
717: } // Ending IRing > 5
718:
719: // Handle IRings 1 thru 5.
720:
721: else {
722:
723:     if(OpReadOutLines & OP_ADDRDBL) {
724:
725:         if(IRing -> State() == 1) {
```

```
726:             A_AR -> Reset();
727:             B_AR -> Reset();
728:             C_AR -> Reset();
729:             D_AR -> Reset();
730:         }
731:
732:         b = B_Reg -> Get();
733:         b = b & BIT_NUM;
734:         b.SetOddParity();
735:
736:         A_AR -> Set(TWOOF5(b), IRing -> State());
737:         B_AR -> Set(TWOOF5(b), IRing -> State());
738:         C_AR -> Set(TWOOF5(b), IRing -> State());
739:         D_AR -> Set(TWOOF5(b), IRing -> State());
740:
741:         if(IRing -> State() == I_RING_5 &&
742:             IndexLatches > 0) {
743:             InstructionIndexStart();           // Start up indexing
744:             return;
745:         }
746:
747:         // Fall thru to step "C"
748:     }
749:
750: // This coding varies slightly from the flowchart on page 45.
751: // The logic is the same, but we test for SAR (opcode G) first
752: // to avoid redundant code.
753:
754: // It works alot easier this way because SAR has two special features:
755: // It doesn't reset AAR (so you can store AAR), and it cannot be
756: // indexed.
757:
758: else if((Op_Reg -> Get() & 0x3f) == OP_SAR_G) {           // SAR
759:
760:     if(IRing -> State() == I_RING_1) {
761:         C_AR -> Reset();
762:     }
763:
764:     b = B_Reg -> Get();
765:     b = b & BIT_NUM;
766:     b.SetOddParity();
767:
768:     C_AR -> Set(TWOOF5(b), IRing -> State());
769: }
770:
771: else {           // Not SAR
772:
773:     if(IRing -> State() == I_RING_1) {
774:         A_AR -> Reset();
775:         C_AR -> Reset();
776:     }
777:
778:     b = B_Reg -> Get();
779:     b = b & BIT_NUM;
780:     b.SetOddParity();
781:
782:     A_AR -> Set(TWOOF5(b), IRing -> State());
783:     C_AR -> Set(TWOOF5(b), IRing -> State());
784:
785:     if(IRing -> State() == I_RING_5 &&
786:         IndexLatches != 0) {
787:         InstructionIndexStart();       // Start up indexing
788:         return;
789:     }
790:
791: }
```

792:  
793: } // End of I Ring 1 - 5  
794:  
795:  
796: // Entry point "C" - address validity checking.  
797: // Odd that this happens in the flow chart \*after\* setting the character  
798: // into the address register. Oh well - the 2-of-5 translater is robust!  
799: // It sets invalid entries to 0.  
800:  
801: // First, handle special characters. They are only allowed on I/O  
802: // (Percent Type) opcodes  
803:  
804: if(B\_Reg -> Get().GetType() == BCD\_SC)  
805: if((OpReadOutLines & OP\_PERCENTTYPE) == 0) { *Check WJr3*  
806: AddressChannelCheck ->  
807: SetStop("Address Channel Check: Special Chars, not % type op");  
808: return;  
809: } *2 of 5 class*  
810: }  
811:  
812: // Next, check for zones, which are only valid at certain times  
813:  
814: if(B\_Reg -> Get().ToInt() & BIT\_ZONE) {  
815: if(!IRingZoneTable[IRing -> State()]) {  
816: AddressChannelCheck ->  
817: SetStop("Address ~~WJr3~~ Check: Zones at invalid IRing time");  
818: return;  
819: }  
820:  
821: // Set the index latches  
822:  
823: IndexLatches |= (B\_Reg -> Get().ToInt() & BIT\_ZONE) >>  
824: ((IRing -> State() == I\_RING\_3 || IRing -> State() == I\_RING\_8)  
825: ? 2 : 4);  
826:  
827: }  
828:  
829: // Step "B": Read out next character, advance I-Ring  
830: // Will pick up again at step "D"  
831:  
832: InstructionDecodeIARAdvance();  
833: return;  
834: }  
835:  
836: // Routine to implement Step "B" in the chart. It takes an I cycle  
837: // (reads out character pointed to by I\_AR), advances IRing, and, if  
838: // the newly read character doesn't have a word mark, advances I\_AR.  
839:  
840: // In the real 1410, what happens with the I\_AR is that the WM inhibits  
841: // setting the I\_AR from STAR+1, and then later causes I\_AR to be copied  
842: // from the STAR (which has the old address).  
843:  
844: void T1410CPU::InstructionDecodeIARAdvance()  
845: {  
846: \*STAR = \*I\_AR; .  
847: IRing -> Next();  
848: Readout();  
849: Cycle();  
850: if(!B\_Reg -> Get().TestWM()) {  
851: I\_AR -> Set(STARMod(+1));  
852: }  
853: return;  
854: }  
855:  
856: // Routine to start up indexing.  
857: // When an index is present at IRing 5 or IRing 11 times, we do this instead

```

858: // of advancing the IRing. This then causes X Cycles (see below)
859:
860: void T1410CPU::InstructionIndexStart()
861: {
862:     CycleRing -> Set(CYCLE_X); // Doing X cycles
863:     ARing -> Reset(); // Reset A Ring to initial state
864:
865:     assert(IndexLatches > 0 && IndexLatches < 16);
866:
867:     // Use the "address generator" to address the proper index register
868:
869:     STAR -> Set(IndexRegisterLookup[IndexLatches]);
870:
871:     // Advance to A2, and read out first index register character
872:     // Address modification by -1 for this cycle
873:
874:     ARing -> Next();
875:     Readout();
876:     Cycle(); // Does nothing
877:     STAR -> Set(STARMod(-1));
878: }
879:
880: // Indexing routine
881:
882: void T1410CPU::InstructionIndex()
883: {
884:     BCD sum;
885:
886:     assert(IRing -> State() == I_RING_5 || IRing -> State() == I_RING_10);
887:
888:     if(IRing -> State() == I_RING_5) {
889:         if(ARing -> State() == A_RING_2) {
890:             if(OpReadOutLines & OP_ADDRDBL) {
891:                 B_AR -> Reset();
892:                 D_AR -> Reset();
893:             }
894:             else {
895:                 ← A_AR -> Reset();
896:             }
897:         }
898:
899:         A_Reg -> Set(C_AR -> GateBCD(6 - (ARing -> State()))); // Gate C Address reg.
900:     }
901:     else {
902:         if(ARing -> State() == A_RING_2) {
903:             B_AR -> Reset();
904:         }
905:
906:         A_Reg -> Set(D_AR -> GateBCD(6 - (ARing -> State()))); // Gate D Address reg.
907:     }
908:
909:     // Determine sign of indexing. If minus, set up a complement add
910:     // (Complement add also requires carry to be set).
911:
912:     if(ARing -> State() == A_RING_2) {
913:         if(B_Reg -> Get().IsMinus()) {
914:             BComplement -> Set();
915:             CarryIn -> Set();
916:         }
917:         else {
918:             CarryIn -> Reset();
919:             BComplement -> Reset();
920:         }
921:     }
922:     else {
923:         CarryIn -> Set(CarryOut -> State());

```

*Should always reset A!  
at A ring 2*

```

924:     }
925:
926:     // Run it thru the adder, with the A channel coming from A Register
927:
928:     Adder(AChannel -> Select(AChannel -> A_Channel_A), false,
929:           B_Reg -> Get(), BComplement -> State());
930:
931:     DEBUG("Indexing. Added %x", AChannel -> Select().ToInt());
932:     DEBUG("And    %x", B_Reg -> Get().ToInt());
933:     DEBUG("COMP   %x", BComplement -> State());
934:     DEBUG("SUM    %x", AdderResult.ToInt());
935:
936:     // Gate adder numerics to assembly channel
937:
938:     sum = AssemblyChannel -> Select(
939:         AssemblyChannel -> AsmChannelWMNone,
940:         AssemblyChannel -> AsmChannelZonesNone,
941:         false,
942:         AssemblyChannel -> AsmChannelSignNone,
943:         AssemblyChannel -> AsmChannelNumAdder );
944:
945:     if(IRing -> State() == 5) {
946:         if(OpReadOutLines & OP_ADDRDBL) {
947:             B_AR -> Set(TWOOF5(sum), 6 - (ARing -> State()));
948:         }
949:         A_AR -> Set(TWOOF5(sum), 6 - (ARing -> State()));
950:     } else {
951:         B_AR -> Set(TWOOF5(sum), 6 - (ARing -> State()));
952:         D_AR -> Set(TWOOF5(sum), 6 - (ARing -> State()));
953:     }
954:
955:
956:     // If indexing operation is done, set registers right, and restart
957:     // the instruction readout process
958:
959:     if(ARing -> State() == A_RING_6) {
960:         if(IRing -> State() == I_RING_5) {
961:             *C_AR = *A_AR;
962:         }
963:         else {
964:             *D_AR = *B_AR;
965:         }
966:         CycleRing -> Set(CYCLE_I); // Doing I cycles again
967:         InstructionDecodeIARAdvance();
968:         return;
969:     }
970:
971:     // Otherwise, advance to the next X Cycle
972:
973:     ARing -> Next();
974:     Readout();
975:     Cycle();
976:     STAR -> Set(STARMod(-1));
977: }
978:
979: // Dummy instruction execution routine: used for invalid/unimplemented ops
980:
981: void T1410CPU::InstructionExecuteInvalid()
982: {
983:     InstructionCheck -> SetStop("Attempt to execute Invalid/Unimplemented Op");
984:     return;
985: }

```

~~A~~ D as well!  
 C & INST REG  
 System Fund.  
 P) 6P

```
1: //-----  
2: #ifndef UI1403H  
3: #define UI1403H  
4: //-----  
5: #include <Classes.hpp>  
6: #include <Controls.hpp>  
7: #include <StdCtrls.hpp>  
8: #include <Forms.hpp>  
9: #include <Buttons.hpp>  
10: #include <Dialogs.hpp>  
11: //-----  
12:  
13: #define PRINTPOSITIONS 132  
14: #define PRINTMAXLINES 1500  
15:  
16: class TFI1403 : public TForm  
17: {  
18:     __published: // IDE-managed Components  
19:         TMemo *Paper;  
20:         TBitBtn *CheckReset;  
21:         TBitBtn *Start;  
22:         TBitBtn *Stop;  
23:         TBitBtn *Space;  
24:         TBitBtn *CarriageRestore;  
25:         TBitBtn *SingleCycle;  
26:         TLabel *LightPrintReady;  
27:         TLabel *LightPrintCheck;  
28:         TLabel *LightEndofForms;  
29:         TBitBtn *CarriageStop;  
30:         TLabel *LightFormsCheck;  
31:         TLabel *LightSyncCheck;  
32:         TButton *FileButton;  
33:         TButton *EnableFile;  
34:         TButton *Button1;  
35:         TButton *EnablePrinter;  
36:         TOpenDialog *FileCaptureDialog;  
37:         TButton *CarriageTape;  
38:         void __fastcall StartClick(TObject *Sender);  
39:         void __fastcall CheckResetClick(TObject *Sender);  
40:         void __fastcall StopClick(TObject *Sender);  
41:         void __fastcall SpaceClick(TObject *Sender);  
42:         void __fastcall CarriageRestoreClick(TObject *Sender);  
43:         void __fastcall CarriageStopClick(TObject *Sender);  
44:         void __fastcall FileButtonClick(TObject *Sender);  
45:  
46:         void __fastcall EnableFileClick(TObject *Sender);  
47:         void __fastcall CarriageTapeClick(TObject *Sender);  
48:     private: // User declarations  
49:  
50:         int PrintPosition;  
51:         char PrintBuffer[PRINTPOSITIONS + 1];  
52:         int Line;  
53:  
54:     public: // User declarations  
55:         __fastcall TFI1403(TComponent* Owner);  
56:         bool SendBCD(BCD c);  
57:         ✓bool DoPrint(); End of Line  
58:         bool NextLine();  
59:  
60:         TPrinter *PrinterIODevice;  
61:  
62: };  
63: //-----  
64: extern PACKAGE TFI1403 *FI1403;  
65: //-----  
66: #endif
```

✓bool PrintData;

```

1: //-----
2: #include <vcl.h>
3: #pragma hdrstop
4:
5: #include <dir.h>
6: #include <stdio.h>
7:
8: #include "UBCD.H"
9: #include "UI1410CPUT.H"
10: #include "UI1410CHANNEL.H"
11: #include "UIPRINTER.h"
12: #include "UI1403.h"
13: //-----
14: #pragma package(smart_init)
15: #pragma resource "*.*dfm"
16:
17: #include "UI1410DEBUG.H"
18:
19: TFI1403 *FI1403;
20: //-----
21: __fastcall TFI1403::TFI1403(TComponent* Owner)
22:   : TForm(Owner)
23: {
24:   Width = 699;
25:   Left = 260;
26:   Top = 0;
27:   Height = 296;
28:   PrintPosition = 0;
29:   Line = 1;
30:
31:   WindowState = wsMinimized;
32: }
33: //-----
34:
35: bool TFI1403::SendBCD(BCD c) {
36:
37:   if(PrintPosition < 0 || PrintPosition >= PRINTPOSITIONS) {
38:     return(false);
39:   }
40:   PrintBuffer[PrintPosition++] = c.ToAscii();
41:   PrintBuffer[PrintPosition] = '\0';
42:   return(true);
43: } End of Line
44:
45: bool TFI1403::DoPrint() {
46:
47:   if(!PrintPosition) { PrintData = true;
48:     return(true);
49:   }
50:   if(Paper -> Lines -> Capacity > PRINTMAXLINES) {
51:     Paper -> Lines -> Delete(0);
52:   }
53:   Paper -> Lines -> Add(PrintBuffer);
54:   ++Line;
55:   PrintPosition = 0;
56:   return(true);
57: }
58:
59: bool TFI1403::NextLine() {
60:
61:   If(PrintPosition)
62:     return(DoPrint());
63:
64:   else {
65:     PrintPosition = 1;
66:   }

```

*PrintData = false;*

*if (!PrintData) {*

*sprintf(PrintBuffer, "<%d>", Line);*

*}*

*Print Position = 0;*

*PrintData = false;*

```
✓ 67:         sprintf(PrintBuffer,"<d>,%d",Line);
68:     }
69: }
70: }
71:
72: void __fastcall TFI1403::StartClick(TObject *Sender)
73: {
74:     if(LightPrintCheck -> Enabled || LightEndofForms -> Enabled ||
75:         LightFormsCheck -> Enabled || LightSyncCheck -> Enabled) {
76:         return;
77:     }
78:     PrinterIODevice -> Start();                                // If OK, will light READY
79: }
80: //-----
81:
82: void __fastcall TFI1403::CheckResetClick(TObject *Sender)
83: {
84:     LightPrintCheck -> Enabled = false;
85:     LightEndofForms -> Enabled = false;
86:     LightFormsCheck -> Enabled = false;
87:     LightSyncCheck -> Enabled = false;
88:     PrinterIODevice -> CheckReset();
89: }
90: //-----
91:
92: void __fastcall TFI1403::StopClick(TObject *Sender)
93: {
94:     PrinterIODevice -> Stop();                                 // Will unlight READY
95: }
96: //-----
97:
98: void __fastcall TFI1403::SpaceClick(TObject *Sender)
99: {
100:     if(PrinterIODevice -> IsReady()) {
101:         return;
102:     }
103:     PrinterIODevice -> CarriageSpace();
104: }
105: //-----
106:
107: void __fastcall TFI1403::CarriageRestoreClick(TObject *Sender)
108: {
109:     if(PrinterIODevice -> IsReady()) {
110:         return;
111:     }
112:     PrinterIODevice -> CarriageRestore();
113: }
114: //-----
115:
116: void __fastcall TFI1403::CarriageStopClick(TObject *Sender)
117: {
118:     PrinterIODevice -> CarriageStop();
119: }
120: //-----
121:
122: void __fastcall TFI1403::FileButtonClick(TObject *Sender)
123: {
124:
125:     // If the button says "Disable", we have to close out the existing file.
126:
127:     if(EnableFile -> Enabled &&
128:         strcmp(EnableFile -> Caption.c_str(),"Close") == 0) {
129:         PrinterIODevice -> FileCaptureClose();
130:         EnableFile -> Caption = "Enable";
131:         EnableFile -> Enabled = false;
132:     }
```

```
133: // Now, send the file name off to the printer to have and to hold
134: if(FileCaptureDialog -> Execute() &&
135:     PrinterIODevice -> FileCaptureSet(FileCaptureDialog -> FileName.c_str())) {
136:     EnableFile -> Caption = "Enable";
137:     EnableFile -> Enabled = true;
138: }
139: }
140: }
141: }
142: //-----
143:
144:
145: void __fastcall TFI1403::EnableFileClick(TObject *Sender)
146: {
147:     if(strcmp(EnableFile -> Caption.c_str(),"Close") == 0) {
148:         PrinterIODevice -> FileCaptureClose();
149:         EnableFile -> Caption = "Enable";
150:     }
151:     else if(PrinterIODevice -> FileCaptureOpen()) {
152:         EnableFile -> Caption = "Close";
153:     }
154: }
155: //-----
156:
157: void __fastcall TFI1403::CarriageTapeClick(TObject *Sender)
158: {
159:     int rc;
160:
161:     if(FileCaptureDialog -> Execute()) {
162:         rc = PrinterIODevice ->
163:             SetCarriageTape(FileCaptureDialog -> FileName.c_str());
164:         if(rc < 0) {
165:             DEBUG("Carriage Tape File Error, line %d",-rc);
166:         }
167:     }
168: }
169: //-----
170:
```

```

1: //-----
2: #ifndef UIPRINTERH
3: #define UIPRINTERH
4: //-----
5:
6: #define PRINTMAXFORM      1024
7: #define PRINTMAXTOKENS    80
8: #define PRINTCCMAXLINE   256
9:
10: #define PRINTER_IO_DEVICE 2
11:
12: //  Printer Adapter Unit (1414)
13:
14: class TPrinter : public T1410IODevice {
15:
16: protected:
17:
18:     int PrintStatus;           // Printer Status for Channel
19:     bool Ready;               // True if ready to go
20:     bool CarriageCheck;       // True if runaway forms
21:     TBusyDevice *BusyEntry;   // Used to set delays
22:     int BufferPosition;       // Current output column
23:     int SkipLines;           // Deferred Skip in lines
24:     int SkipChannel;          // Deferred Skip to Channel
25:
26:     int FormLength;           // Length of current form
27:     int FormLine;              // Current line in form
28:
29:     TFileStream *ccfd;         // Carriage Control File
30:     int CarriageTape[PRINTMAXFORM]; // Carriage tape data
31:     char ccline[PRINTCCMAXLINE];
32:
33:     char FileName[MAXPATH];   // File name, if to file
34:     TFileStream *fd;           // File FDs for print, CC file
35:
36: public:
37:
38:     TPrinter(int devicenum, T1410Channel *Channel);
39:
40:     inline bool IsReady() { return(Ready); }
41:     inline bool IsBusy() { return BusyEntry -> TestBusy(); }
42:
43:     void Start();                // Called from UI Start Button
44:     void Stop();                 // Called from UI Stop Button
45:     void CheckReset();           // Called from UI Check Reset
46:
47:     bool CarriageRestore();      // Carriage to Channel 1
48:     void CarriageStop();         // Called from UI Carriage Stop
49:     bool CarriageSpace();        // Space to next line
50:
51:     virtual int Select();        // Channel Select
52:     virtual void DoOutput();     // Character to output
53:     virtual void DoInput();      // NOP on this device
54:     virtual int StatusSample();  // End of I/O status sample
55:     virtual void DoUnitControl(BCD opmod); // CC operation
56:
57:     void ControlCarriage(BCD opmod);
58:
59:     void DoOutputChar(BCD c);   // Send one char of output
60:     void DoPrint(); End of Line(); // Flush line to device.
61:     // TODO bool PrintAscii(char c); // Output to a file or printer
62:
63:     int SetCarriageTape(char *tapefile); // Set up carriage control
64:     void SetCarriageDefault();          // Set default CC tape
65:     bool CarriageChannelTest(int cchannel); // True if this line has chan.
66:

```

*bool CarriageAdvance;*

```
67:     void FileCaptureClose();           // Terminate file capture
68:     bool FileCaptureSet(char *filename); // Set capture file name
69:     bool FileCaptureOpen();            // Open capture filename
70:     bool FileCapturePrint(char c);    // Print a character to file
71:
72: private:
73:
74:     bool CarriageSkip(int lines, int channel);
75:     bool GetCarriageLine();
76:     void ParseCarriageLine(char *line, char **elements);
77:     int CarriageTapeError(int rc);      // Cleans up after tape errors
78:
79: };
80:
81: #endif
```

```
1: //-----
2: #include <vcl.h>
3: #pragma hdrstop
4:
5: #include <dir.h>
6: #include <assert.h>
7: #include <stdlib.h>
8: #include <stdio.h>
9: #include <ctype.h>
10: #include "UBCD.H"
11: #include "UI1410CPU.T"
12: #include "UI1410CHANNEL.H"
13: #include "UIPRINTER.H"
14:
15: //-----
16: #pragma package(smart_init)
17:
18: #include "UI1403.H"
19: #include "UI1410DEBUG.H"
20:
21: // Printer Adapter Unit Implementation. Follows I/O Device Interface.
22:
23: // Constructor. Creates a printer! (NOTE: RIGHT NOW, CAN ONLY BE 1 !!)
24:
25: TPrinter::TPrinter(int devicenum, T1410Channel *Channel) :
26:     T1410IODevice(devicenum, Channel) {
27:
28:     BusyEntry = new TBusyDevice();
29:
30:     ccfd = NULL;
31:     ccline[0] = '\0';
32:     SetCarriageDefault();                                // Default CC tape
33:
34:     FileName[0] = '\0';
35:     fd = NULL;
36:
37:     SkipLines = SkipChannel = 0;
38:     BufferPosition = 0;
39:
40:     PrintStatus = 0;
41:     Start();
42: }
43:
44: // Start. Called from initialization and when Start button on printer
45: // user interface is pressed.
46:
47: void TPrinter::Start() {
48:     if(CarriageCheck) {
49:         return;
50:     }
51:     Ready = true;
52:     FI1403 -> LightPrintReady -> Enabled = true;
53: }
54:
55: // Stop. Called when Stop button on printer user interface is pressed.
56:
57: void TPrinter::Stop() {
58:     Ready = false;
59:     FI1403 -> LightPrintReady -> Enabled = false;
60: }
61:
62: // CheckReset: You guessed it: Called when the Check Reset button on the
63: // printer user interface is pressed.
64:
65: void TPrinter::CheckReset() {
66:     CarriageCheck = false;
```

Carriage Advance = false;

```
67: }
68:
69: // Carriage Restore: You know the routine.
70:
71: bool TPrinter::CarriageRestore() {
72:     return(CarriageSkip(0,1));
73: }
74:
75: // Carriage Space. This one does a little more, because it is also
76: // called during Carriage Skip operations.
77:
78: bool TPrinter::CarriageSpace() {
79:
80:     bool status;
81:
82:     status = FI1403 -> NextLine();
83:     if(!status) {
84:         PrintStatus |= IOCHCONDITION;
85:     }
86:
87:     if(fd != NULL) {
88:         if(!FileCapturePrint('\r') || !FileCapturePrint('\n')) {
89:             PrintStatus |= IOCHCONDITION;
90:             status = false;
91:         }
92:     }
93:
94:     if(++FormLine > FormLength) {
95:         FormLine = 1;
96:     }
97:     return(status);
98: }
99:
100: // Carriage Stop
101:
102: void TPrinter::CarriageStop() {
103:     Stop();
104:     CarriageCheck = true;
105:     FI1403 -> LightFormsCheck -> Enabled = true;
106: }
107:
108: // Method to assign a name to the capture file
109:
110: bool TPrinter::FileCaptureSet(char *filename) {
111:     if(strlen(filename) >= MAXPATH) {
112:         return(false);
113:     }
114:     strcpy(fileName,filename);
115:     return(true);
116: }
117:
118: // And, finally, open the capture file...
119:
120: bool TPrinter::FileCaptureOpen() {
121:     if(strlen(fileName) == 0) {
122:         return(false);
123:     }
124:     fd = new TFileStream(fileName,fmCreate | fmOpenWrite);
125:     return(fd != NULL);
126: }
127:
128:
129: // Method to close down the capture file.
130:
131: void TPrinter::FileCaptureClose() {
132:     if(fd != NULL) {  

CarriageAdvance = false;
```

```
133:         delete fd;
134:         fd = NULL;
135:     }
136: }
137:
138: // Now, for the I/O Device Interface.
139:
140: // Select is called at the beginning of an IO operation.
141:
142: int TPrinter::Select() {
143:
144:     if(!Ready) {
145:         return(IOCNOTREADY);
146:     }
147:     if(BusyEntry -> TestBusy()) {
148:         return(IOCMBUSY);
149:     }
150:     BufferPosition = 0;
151:     if(Channel -> ChRead -> State() ||
152:         (Channel -> GetUnitNumber() != 0 && Channel -> GetUnitNumber() != 1) ) {
153:         return(IOCNOTREADY);
154:     }
155:     return(PrintStatus = 0);
156: }
157:
158: // Status Sample is called at the end of an I/O operation.
159:
160: int TPrinter::StatusSample() {
161:     if(Channel -> ChWrite -> State() && BufferPosition != PRINTPOSITIONS) {
162:         PrintStatus |= IOCHWLRECORD;
163:     }
164:     return(Channel -> GetStatus() | PrintStatus);
165: }
166:
167: // Here is where the real work gets done...
168:
169: void TPrinter::DoOutput() {
170:
171:     BCD ch_char;
172:
173:     // If not ready, say so.
174:
175:     if(!Ready) {
176:         PrintStatus |= IOCNOTREADY;
177:         Channel -> ExtEndofTransfer = true;
178:         return;
179:     }
180:
181:     // Get character from the channel silo
182:
183:     ch_char = Channel -> ChR2 -> Get();
184:
185:     // If we have too many characters, indicate the problem
186:
187:     if(BufferPosition >= PRINTPOSITIONS) {
188:         PrintStatus |= IOCHWLRECORD;
189:         Channel -> ExtEndofTransfer = true;
190:         return;
191:     }
192:
193:     // Check the parity of the character coming from memory
194:
195:     if(!ch_char.CheckParity()) {
196:         PrintStatus |= IOCHDATACHECK;
197:     }
198:
```

if (Carry Advance){  
 Carriage Spew();  
}

```
199: // In Load Mode, we turn Wordmarks into Word Separators.
200: // The addition of the word separator makes a Wrong Length Record very
201: // very likely!
202:
203: if(Channel -> LoadMode && ch_char.TestWM()) {
204:     DoOutputChar(BCD_WS);
205:     if(++BufferPosition >= PRINTPOSITIONS) {
206:         PrintStatus |= IOCHWLRECORD;
207:         Channel -> ExtEndofTransfer;
208:         return;
209:     }
210: }
211:
212: // Having handled Load Mode, we proceed. For the normal print unit (%20),
213: // we now print the character. If we are in Move Mode and have the
214: // alternate print unit (%21), we print a space, because we have already
215: // stripped the WM. (Not sure this is how it really worked, but the
216: // Principles of Operation indicates that a L%21 prints a blank line).
217:
218: if(Channel -> GetUnitNumber() == 0) {
219:     DoOutputChar(ch_char);
220: }
221: else { // %21
222:     DoOutputChar(Channel -> MoveMode && ch_char.TestWM() ?
223:                     BCD_1 : BCD_SPACE);
224: }
225: ++BufferPosition;
226:
227: // Check for end of transfer. If we have need more characters,
228: // let the Channel know. Otherwise, let the Channel know we have
229: // had enough.
230:
231: // Also, this is normally how we will know we are ready to print the
232: // line. (A Wrong Length Record can suppress output, according to the
233: // Principles of Operation). It is easier to do it here than wait to
234: // do it a StatusSample time - which has to service both Print and
235: // Carriage Control operations. After we print the line, we might
236: // also have a deferred Carriage Control operation to do.
237:
238: if(BufferPosition == PRINTPOSITIONS) {
239:     ✓ DoPrint(); EndOfLine();
240:     if(SkipLines || SkipChannel) {
241:         CarriageSkip(SkipLines, SkipChannel);
242:         SkipLines = SkipChannel = 0;
243:     }
244:     else {
245:         ✓ carriageSpace();
246:     }
247:     BusyEntry -> SetBusy(2);
248: }
249:
250: // Minor kludge. We ask for another character, even if we just
251: // printed. If there is one, then we will detect WLR.
252:
253: Channel -> OutputRequest = true;
254: Channel -> CycleRequired = true;
255: }
256:
257: // Send the output character to the appropriate device.
258:
259: void TPrinter::DoOutputChar(BCD c) {
260:
261:     c = c & (BIT_NUM | BIT_ZONE);
262:     if(!FI1403 -> SendBCD(c)) {
263:         PrintStatus |= IOCHCONDITION;
264:     }
```

Carriage Advance = true;

```

265:     if(fd != NULL) {
266:         if(!FileCapturePrint(c.ToAscii())) {
267:             PrintStatus |= IOCHCONDITION;
268:         }
269:     }
270: }
271:
272: // Send the word to the appropriate device to actually print the line.
273: End of Line
274: void TPrinter::DoPrint() {
275:     End of Line
276:     if(!FI1403 -> DoPrint()) {
277:         PrintStatus |= IOCHCONDITION;
278:     }
279:     // File Capture does nothing here...
280: }
281:
282: }
283:
284: bool TPrinter::FileCapturePrint(char c) {
285:
286:     if(fd == NULL) {
287:         return(false);
288:     }
289:     if(fd -> Write(&c,1) != 1) {
290:         DEBUG("TPrinter::FileCapturePrint Error",0);
291:         return(false);
292:     }
293:     return(true);
294: }
295:
296: // Carriage Control operations. Basically pretty simple stuff.
297:
298: void TPrinter::DoUnitControl(BCD opmod) {
299:
300:     Channel -> UnitControlOverlapBusy = NULL;
301:     if(!Ready) {
302:         PrintStatus |= IOCHNOTREADY;
303:     }
304:     else {
305:         ControlCarriage(opmod);
306:         BusyEntry -> SetBusy(2);
307:     }
308:     Channel -> ExtEndofTransfer = true;
309: }
310:
311: void TPrinter::ControlCarriage(BCD opmod) {
312:
313:     int dint;
314:     int num;
315:
316:     dint = opmod.To6Bit();
317:     num = dint & BIT_NUM;
318:
319:     // Single spaces get handled by default (may change later)
320:
321:     if(num == 0) {
322:         return;
323:     }
324:     SkipChannel = SkipLines = 0;
325:
326:
327:     // 0 - 9, #, @: Skip to carriage tape channels 1 - 12
328:
329:     if((dint & BIT_ZONE) == 0) {
330:         if(num > 12) {
}

```

*if (fd != NULL &&*  
*!FileCapturePrint('r')) {*  
*PrintStatus |= IOCHCONDITION;*  
*}*

*if (dint == opmod.SymbolMode) {*  
*Carriage Advance = false;*  
*}*

```
331:         return;
332:     }
333:     CarriageSkip(0,num);
334: }
335:
336: // A -I, ? . <lozenge>: Skip to channels 1-2 *after* printing
337:
338: else if((dint & BIT_ZONE) == BIT_ZONE) {
339:     if(num <= 12) {
340:         SkipChannel = num;
341:     }
342: }
343:
344: // J, K, L: Immediate space of 1, 2 or 3 lines
345:
346: else if((dint & BIT_ZONE) == BITB) {
347:     if(num > 3) {
348:         return;
349:     }
350:     CarriageSkip(num,0);
351: }
352:
353: // /, S, T: Skip 1, 2 or 3 lines *after* printing
354:
355: else {
356:     if(num > 3) {
357:         return;
358:     }
359:     SkipLines = num;
360: }
361: }
362:
363: // Printers ignore input. (Actually, this should never get called, because
364: // select will return Not Ready on an attempt to select for input).
365:
366: void TPrinter::DoInput() {
367:     PrintStatus |= IOCHNOTREADY;
368:     return;
369: }
370:
371: // PRIVATE routine to skip carriage to given channel or given number of lines.
372:
373: bool TPrinter::CarriageSkip(int spaces, int chan) {
374:
375:     int num;
376:     int line;
377:
378:     if(spaces < 0 || spaces > 4 || chan < 0 || chan > 12) {
379:         CarriageStop();
380:         return(false);
381:     }
382:
383:     if(spaces > 0 && spaces < 4) {
384:         while(spaces--) {
385:             CarriageSpace();
386:         }
387:         return(true);
388:     }
389:
390:     if(chan == 0) {
391:         return(true);
392:     }
393:
394:     num = (1 << (chan - 1)),
395:
396:     for(line=0; line <= FormLength; ++line) {
```

```
397:         CarriageSpace();
398:         if(CarriageChannelTest(num) {
399:             return(true);    chan
400:         }
401:     }
402:
403:     CarriageStop();
404:     return(false);
405: }
406:
407: bool TPrinter::CarriageChannelTest(int ch) {
408:     return((CarriageTape[FormLine] & (1 << (ch-1))) != 0);
409: }
410:
411: // The remaining code has to do with setting up the carriage control tape.
412: // A carriage control tape is read in from a file using the same format
413: // as the Newcomer 1401 simulator.
414:
415: // * beginning a line indicates a comment
416: // LENGTH ###
417: // CHAN # = # [+/#] ...
418:
419: // Method to set up a default carriage tape, during construction
420:
421: void TPrinter::SetCarriageDefault() {
422:
423:     int line;
424:
425:     for(line=0; line < PRINTMAXFORM; ++line) {
426:         CarriageTape[line] = 0;
427:     }
428:
429:     FormLength = 66;
430:     FormLine = 1;
431:     CarriageTape[4] = 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 512 | 1024;
432:     // Channesl      1   2   3   4   5   6   7   8   10   11
433:
434:     CarriageTape[61] = 256;      // Channel 9
435:     CarriageTape[63] = 2048;    // Channel 12
436: }
437:
438: // Main method for carriage tape - sets up carriage tape from a file
439: // Returns 0 if tape is OK. Returns - value if an error. The value
440: // is in fact the line number.
441:
442: int TPrinter::SetCarriageTape(char *filename) {
443:
444:     char *elements[PRINTMAXTOKENS];           // Up to 80 fields per line.
445:     int line, chan, i, n;
446:
447:     for(line=0; line < PRINTMAXFORM; ++line) {
448:         CarriageTape[line] = 0;
449:     }
450:
451:     // Passing us no file is OK: Means they want the default carriage tape.
452:
453:     if(filename == NULL || strlen(filename) == 0) {
454:         SetCarriageDefault();
455:         return(0);
456:     }
457:
458:     assert(ccfd == NULL);
459:     line = 0;
460:
461:     // Try and open the file. If it fails, report an error on line 1.
462:
```

```
463:     try {
464:         ccfд = new TFileStream(filename, fmOpenRead);
465:     }
466:     catch(EFOpenError &e) {
467:         return(CarriageTapeError(-1));
468:     }
469:
470: // Skip any leading comment lines. Error if we hit EOF.
471:
472: do {
473:     if(!GetCarriageLine()) {
474:         return(CarriageTapeError(-line));
475:     }
476:     ++line;
477: } while(ccline[0] == '*');
478:
479: // Parse the line. It should have 2 elements. LENGTH and the number
480:
481: ParseCarriageLine(ccline,elements);
482: if(strcmp(elements[0],"LENGTH") != 0) {
483:     return(CarriageTapeError(-line));
484: }
485: FormLength = atoi(elements[1]);
486: if(FormLength < 1 || FormLength > PRINTMAXFORM) {
487:     return(CarriageTapeError(-line));
488: }
489:
490: // Now process the carriage control lines themselves.
491:
492: n = 0;
493: while(GetCarriageLine()) {
494:     ++line;
495:     if(ccline[0] == '*') {                                // * for comment
496:         continue;
497:     }
498:     ParseCarriageLine(ccline,elements);
499:     if(strcmp(elements[0],"CHAN") != 0 ||           // Check CHAN #
500:         strcmp(elements[2],"=") != 0) {
501:         return(CarriageTapeError(-line));
502:     }
503:     chan = atoi(elements[1]);
504:     if(chan < 1 || chan > 12) {                      // Validate channel number
505:         return(CarriageTapeError(-line));
506:     }
507:     n = 0;
508:     for(i=3; elements[i] != NULL; ++i) {            // Process the form lines
509:         if(*elements[i] == '+') {
510:             n += atoi(elements[i]+1);
511:         }
512:         else {
513:             n = atoi(elements[i]);
514:         }
515:         if(n < 1 || n > FormLength) {
516:             CarriageTapeError(-line);
517:         }
518:         CarriageTape[n] |= (1 << (chan - 1));
519:     }
520: }
521:
522: FormLine = 1;
523: delete ccfд;
524: ccfд = NULL;
525: return(0);
526: }
527:
528: // Utility method to clean up after carriage tape errors
```

```
529:
530: int TPrinter::CarriageTapeError(int rc) {
531:
532:     if(ccfd != NULL) {
533:         delete ccfд;
534:     }
535:     ccfд = NULL;
536:     SetCarriageDefault();
537:     return(rc);
538: }
539:
540: // Method to get one line of input. (Why Borland didn't have this kind of
541: // method as part of their file stream object I have *no* idea!
542:
543: bool TPrinter::GetCarriageLine() {
544:
545:     char *cp;
546:
547:     if(ccfd == NULL) {
548:         return(false);
549:     }
550:
551:     for(cp = ccline; cp - ccline < PRINTCCMAXLINE; ++cp) {
552:         if(ccfd -> Read(cp,1) != 1) {
553:             return(false);
554:         }
555:         if(*cp == '\n') {
556:             *cp = 0;
557:             return(true);
558:         }
559:     }
560:
561:     ccline[PRINTCCMAXLINE] = 0;
562:     return(false);
563: }
564:
565: // Method to set pointers to the tokens in a line.
566:
567: void TPrinter::ParseCarriageLine(char *line, char **element) {
568:
569:     int i=0;
570:     element[0] = NULL;
571:
572:     // Ignore leading white space to find first token
573:
574:     while(*line && isspace(*line)) {
575:         ++line;
576:     }
577:
578:     // Process tokens one at a time until end of line.
579:
580:     while(*line && i < PRINTMAXTOKENS-1) {
581:         element[i] = line;
582:         element[i+1] = NULL;
583:         while(++line && !isspace(*line)) {
584:             // Skip over rest of the token
585:         }
586:         if(!*line) {
587:             break;
588:         }
589:         *line = 0;                                // Terminate token with '\0'
590:         while(++line && isspace(*line)) {
591:             // Skip over white space
592:         }
593:         ++i;                                     // Bump to next element
594:     }
595:
```

595: }  
596:

C

C

C

```
1: //-----
2: #ifndef UI1402H
3: #define UI1402H
4: //-----
5: #include <Classes.hpp>
6: #include <Controls.hpp>
7: #include <StdCtrls.hpp>
8: #include <Forms.hpp>
9: #include <Buttons.hpp>
10: #include <Dialogs.hpp>
11: //-----
12: class TFI1402 : public TForm
13: {
14:     __published:    // IDE-managed Components
15:     TBitBtn *ReaderStart;
16:     TBitBtn *ReaderStop;
17:     TBitBtn *EOFButton;
18:     TBitBtn *PunchStart;
19:     TBitBtn *PunchStop;
20:     TLabel *LightPunchReady;
21:     TLabel *LightPunchCheck;
22:     TLabel *LightPunchStop;
23:     TLabel *LightChips;
24:     TLabel *LightReaderValidity;
25:     TLabel *LightFuse;
26:     TLabel *LightPower;
27:     TLabel *LightTransport;
28:     TLabel *LightStacker;
29:     TLabel *LightReaderReady;
30:     TLabel *LightReaderCheck;
31:     TLabel *LightReaderStop;
32:     TButton *LoadReaderHopper;
33:     TButton *LoadPunchHopper;
34:     TOpenDialog *FileOpenDialog;
35:     void __fastcall ReaderStartClick(TObject *Sender);
36:     void __fastcall ReaderStopClick(TObject *Sender);
37:     void __fastcall EOFButtonClick(TObject *Sender);
38:     void __fastcall LoadReaderHopperClick(TObject *Sender);
39: private:      // User declarations
40: public:        // User declarations
41:     __fastcall TFI1402(TComponent* Owner);
42:
43:     void ResetEOF();
44:     void SetReaderCheck(bool flag);
45:     void SetReaderValidity(bool flag);
46:     void SetReaderReady(bool flag);
47:     void SetPunchReady(bool flag);
48:
49:     TCardReader *ReaderIODevice;
50: };
51: //-----
52: extern PACKAGE TFI1402 *FI1402;
53: //-----
54: #endif
```

```
1: //-----
2: #include <vcl.h>
3: #pragma hdrstop
4:
5: #include <dir.h>
6: #include "UBCD.H"
7: #include "UI1410CPUT.H"
8: #include "UI1410CHANNEL.H"
9: #include "UIREADER.H"
10: #include "UI1402.h"
11: //-----
12: #pragma package(smart_init)
13: #pragma resource "*.*dfm"
14: TFI1402 *FI1402;
15: //-----
16: __fastcall TFI1402::TFI1402(TComponent* Owner)
17:     : TForm(Owner)
18: {
19:     Width = 632;
20:     Left = 260;
21:     Top = 300;
22:     Height = 164;
23:     WindowState = wsMinimized;
24: }
25: //-----
26:
27: // Method to reset card reader EOF status. In a real 1402, this would
28: // turn out the 1402 light (and enable the button). Here, we just enable
29: // the button.
30:
31: void TFI1402::ResetEOF() {
32:     EOFButton -> Enabled = true;
33: }
34:
35: // Method to set or reset the Reader Check light
36:
37: void TFI1402::SetReaderCheck(bool flag) {
38:     LightReaderCheck -> Enabled = flag;
39: }
40:
41: // Method to set or reset the Reader Validity Light
42:
43: void TFI1402::SetReaderValidity(bool flag) {
44:     LightReaderValidity -> Enabled = flag;
45: }
46:
47: // Method to set or reset the Reader Ready Light
48:
49: void TFI1402::SetReaderReady(bool flag) {
50:     LightReaderReady -> Enabled = flag;
51:     ReaderStart -> Enabled = !flag;
52:     ReaderStop -> Enabled = flag;
53: }
54:
55: // And another to set or reset the Punch Ready light
56:
57: void TFI1402::SetPunchReady(bool flag) {
58:     LightPunchReady -> Enabled = flag;
59:     PunchStart -> Enabled = !flag;
60:     PunchStop -> Enabled = flag;
61: }
62:
63: void __fastcall TFI1402::ReaderStartClick(TObject *Sender)
64: {
65:     if(LightReaderStop -> Enabled) {
66:         return;
```

```
67:     }
68:     if(ReaderIODevice -> DoStart()) {
69:         SetReaderReady(true);
70:     }
71: }
72: //-----
73:
74: void __fastcall TFI1402::ReaderStopClick(TObject *Sender)
75: {
76:     SetReaderReady(false);
77:     EOFButton -> Enabled = true;
78:     ReaderIODevice -> DoStop();
79: }
80: //-----
81:
82: void __fastcall TFI1402::EOFButtonClick(TObject *Sender)
83: {
84:     ReaderIODevice -> SetEOF();
85:     EOFButton -> Enabled = false;
86: }
87: //-----
88:
89: void __fastcall TFI1402::LoadReaderHopperClick(TObject *Sender)
90: {
91:     if(LightReaderReady -> Enabled) {
92:         return;
93:     }
94:     ReaderIODevice -> CloseFile();
95:     if(FileOpenDialog -> Execute() &&
96:         ReaderIODevice -> LoadFile(FileOpenDialog -> FileName.c_str())) {
97:         ReaderStart -> Enabled = true;
98:         EOFButton -> Enabled = true;
99:     }
100: }
101: //-----
102:
```

```

1: //-----
2: #ifndef UIREADERH
3: #define UIREADERH
4: //-----
5:
6: #define READER_IO_DEVICE 1
7:
8: // Very simple class (almost a "struct") to represent a card image
9:
10: class TCard : public TObject {
11:
12: private:
13:     TFileStream *Hopper; THopper *Hopper;
14:     TFileStream *Hopper; THopper *h
15:
16: public:
17:
18:     TCard();                                // Constructor
19:     char image[80];                         // 80 columns of data
20:     void SelectStacker(TFileStream *fd);    // Select Stacker
21:     bool Stack();                           // Stack card to hopper
22: };
23:
24: // Class to implement the card reader interface and buffer
25:
26: class TCardReader : public T1410IODevice {
27:
28: protected:
29:
30:     TCard *ReadStation;                    // Station to read the card
31:     TCard *CheckStation;                  // Where feed goes to
32:     TCard *StackStation;                 // Card after read
33:     TCard *ReadBuffer;                   // Where the read card goes
34:
35:     char filename[MAXPATH];              // Input hopper, if you will
36:     TFileStream *fd;                     // Input hopper file stream
37:
38:     TFileStream *hopperfd[3];          // The 3 card read hoppers
39:
40:     int column;                          // Current column. 1-80, 81
41:
42:     bool ready;                         // True if reader ready
43:     bool eof;                            // True if EOF switch on
44:     bool buffertransferred;             // True if no data at Check Stn
45:
46:     int unit;                            // Hopper number: 0, 1, 2
47:
48:     int readerstatus;                  // Channel status value
49:     TBusyDevice *BusyEntry;
50:
51: public:
52:     // Implement the I/O device interface standard
53:     TCardReader(int devicenum, T1410Channel *Channel);
54:     virtual int Select();
55:     virtual void DoOutput();
56:     virtual void DoInput();
57:     virtual int StatusSample();
58:     virtual void DoUnitControl(BCD opmod);
59:
60:     // State and status methods
61:
62:     inline bool IsReady() { return ready; }
63:     inline void SetEOF() { eof = true; }
64:     inline void ResetEOF() { eof = false; }
65:
66:

```

*Map to  
THopper.cpp*

```
67:     inline bool IsBusy() { return BusyEntry -> TestBusy(); }
68:     inline char *GetFileName() { return filename; }
69:     bool SetUnit(int u);                                // Check unit validity, set
70:     inline int GetUnit() { return unit; }
71:
72:     bool LoadFile(char *s);                            // Call to open input file
73:     void CloseFile();                                 // Force a file close if open
74:     bool DoStart();                                  // Process Start Button
75:     void DoStop();                                   // Process Stop Button
76:
77: private:
78:
79:     void TransportCard(int hopper);                  // Transport card to hopper
80:     TCard *FeedCard();                             // Feed card from input file
81:     int DoInputColumn();                           // Process one card column
82:
83: };
84:
85:
86:
87: #endif
```

```
1: //-----
2: #include <vcl.h>
3: #pragma hdrstop
4:
5: #include <dir.h>
6: #include <stdio.h>
7: #include "UBCD.H"
8: #include "UI1410CPUUT.H"
9: #include "UI1410INST.H"
10: #include "UI1410CHANNEL.H"
11: #include "UIREADER.h"
12:
13: //-----
14: #pragma package(smart_init)
15:
16: #include "UI1410DEBUG.H"
17: #include "UI1402.H"
18:
19: // TCard Class Implementation
20:
21: TCard::TCard() {
22:     Hopper = NULL;
23: }
24:
25: bool TCard::Stack() {
26:     // TODO return (Hopper → Stack (this));
27:     return(true);
28: }
29: T Hopper * h
30: void TCard::SelectStacker(TFileStream *fd) {
31:     Hopper = for h
32: }
33: 
34:
35:
36: // TCardReader Class Implementation
37:
38: // Constructor. Sets up the card reader
39:
40: TCardReader::TCardReader(int devicenum, T1410Channel *Channel) :
41:     T1410IODevice(devicenum, Channel) {
42:
43:     int i;
44:
45:     BusyEntry = new TBusyDevice();
46:
47:     filename[0] = '\0';
48:     fd = NULL;
49:
50:     ReadStation = NULL;
51:     CheckStation = NULL;
52:     StackStation = NULL;
53:     ReadBuffer = NULL;
54:
55:     for(i=0, i < 3, ++i) {
56:         hopperfd[i] = NULL;
57:     }
58:
59:     ready = eof = buffertransferred = false;
60:     readerstatus = column = 0;
61: }
62:
63: // IO Device Implementation
64:
65: // Select. Returns 0 if successful, non 0 channel status otherwise
66:
```

Moved  
To  
UIHOPPER.cpp

```

67: int TCardReader::Select() {
68:
69:     int op;
70:
71:     column = 1;                                // Reset column back to start
72:
73:     // Check that the card reader is ready, not busy, that we are doing
74:     // either a read or unit control operation. If not, return appropriate
75:     // status.
76:
77:     if(!IsReady()) {
78:         return(readerstatus = IOCHNOTREADY);
79:     }
80:     if(IsBusy()) {
81:         return(readerstatus = IOCHBUSY);
82:     }
83:     if(Channel -> ChWrite -> State()) {
84:         return(readerstatus = IOCHNOTREADY);
85:     }
86:
87:     // If this is a read operation (opcode M or L), and there is
88:     // nothing in the read buffer, and eof is set, return IOCHCONDITION.
89:
90:     // If the eof switch is set, we can continue so long as there
91:     // is at least a card at the Read Station.
92:
93:     // If the eof switch is not set, and there is no card at the
94:     // check station, return not ready!
95:
96:     // If that is all OK, call SetUnit to fill the read buffer and start
97:     // the card transport.
98:
99:     // Card transport for SSF is handled in the DoUnitControl method).
100:
101:    readerstatus = 0;
102:
103:    // Nothing more to read, and EOF. Stack the last card, go not ready,
104:    // and return IOCHCONDITION status for M or L opcode.
105:
106:    if(ReadStation == NULL && eof) {
107:        eof = false;
108:        if(StackStation != NULL) {
109:            StackStation -> Stack();           ← { delete StackStation;
110:        }
111:        FI1402 -> ResetEOF();
112:        FI1402 -> SetReaderReady(false);
113:        ready = false;
114:        if((op = CPU -> Op_Reg -> Get()).To6Bit() == OP_IO_MOVE ||
115:            op == OP_IO_LOAD) {
116:                return(readerstatus = IOCHCONDITION);
117:            }
118:            return(readerstatus = IOCHNOTREADY);
119:        }
120:
121:        // If there is no card at the Check Station and the EOF switch isn't on,
122:        // Go not ready, and return not ready.
123:
124:        if(CheckStation == NULL && !eof) {
125:            FI1402 -> SetReaderReady(false);
126:            ready = false;
127:            return(readerstatus = IOCHNOTREADY);
128:        }
129:
130:        if(!SetUnit(Channel -> GetUnitNumber())) {
131:            return(readerstatus = IOCHNOTREADY);
132:        }

```

✓

{ StackStation = NULL;

assert(ReadStation != NULL);

## UIREADER.cpp

```

133:     // Otherwise, AOK. (Might be IOCHCONDITION from SetUnit() ) .
134:     return(readerstatus);
135:
136: }
137:
138:
139: // Card reader unit control - used for Select Stacker and Feed instruction
140:
141: void TCardReader::DoUnitControl(BCD opmod) {
142:
143:     int hopper;
144:
145:     hopper = opmod.To6Bit();
146:
147:     switch(hopper) {
148:
149:     case 0: HOPPER-#1
150:     case 1:   1
151:     case 2:   2
152:         TransportCard(hopper);                                // Move cards along. Stack.
153:         buffertransferred = false;                            // We now have data avail.
154:         Channel -> UnitControlOverlapBusy = BusyEntry; // Be busy for a while
155:         break;
156:
157:     default:
158:         DEBUG("TCardReader::DoUnitControl invalid unit: %d",hopper);
159:         Channel -> UnitControlOverlapBusy = NULL;
160:         readerstatus |= IOCHNOTREADY;
161:         Channel -> ExtEndofTransfer = true;
162:         break;
163:     }
164:
165:     return;
166: }
167:
168: // Card Readers don't do ouptut very well, do they....
169:
170: void TCardReader::DoOutput() {
171:     readerstatus |= IOCHNOTREADY;
172: }
173:
174: // Status sample is pretty simple for this device...
175:
176: int TCardReader::StatusSample() {
177:     return(Channel -> GetStatus() | readerstatus);
178: }
179:
180: // The real meat - input!
181:
182: void TCardReader::DoInput() {
183:
184:     bool wm = false;                                     // True if wm in progress
185:     BCD c;                                              // Column as BCD char
186:     int card_input_char;                               // Column as BCD char too
187:
188:     // If the read buffer is empty, no transfer.
189:     // (really, this should never happen!)
190:
191:     if(ReadBuffer == NULL) {
192:         readerstatus |= IOCHNOTTRANSFER;
193:         Channel -> ExtEndofTransfer = true;
194:         return;
195:     }
196:
197:     // Read the next column. If return value is < 0, then something unusual
198:     // happened. channel status should already be set, so just return.

```

```
199:     card_input_char = DoInputColumn();           // Get next column of data
200:     if(card_input_char < 0) {
201:         return;
202:     }
203: }
204:
205: // Handle load mode
206:
207: if(Channel -> LoadMode && (card_input_char & 0x3f) == (BCD_WS & 0x3f)) {
208:     wm = true;
209:     card_input_char = DoInputColumn();           // Read char after ws
210:     if(card_input_char < 0) {                   // Oops -- off the end
211:         return;
212:     }
213:     if((card_input_char & 0x3f) == (BCD_WS & 0x3f)) { // Another WS?
214:         wm = false;                            // Yes. Throw away WM
215:     }
216: }
217:
218: // Convert the character to BCD. The DoInputColumn routine already
219: // sets IOCHDATACHECK for invalid characters...
220:
221: c = BCD(card_input_char);
222: c.SetOddParity(); ← assert(c >= 0 && c < 64);
223: if(wm) {
224:     c.SetWM();
225:     c.ComplementCheck();
226: }
227:
228: // Tell the channel we have something for it!
229:
230: Channel -> ChannelStrobe(c);
231: }
232:
233: // Method to handle setting the unit during the select process for Read
234:
235: bool TCardReader::SetUnit(int u) {
236:
237:     switch(u) {
238:
239:     case 0: HOPPER-RD
240:     case 1:
241:     case 2: } assert(ReadStation != NULL);
242:     ReadBuffer = ReadStation;
243:     TransportCard(u);
244:     return(true);
245:
246:     case 9:
247:         if(buffertransferred) {
248:             readerstatus |= IOCHNOTTRANSFER;
249:             return(true);
250:         }
251:         ReadBuffer = ReadStation;
252:         return(true);
253:
254:     default:
255:         return(false);
256:     }
257: }
258:
259: // Method to transport the cards in the card reader. The card in the stacking
260: // station gets stacked (and deleted). The card that was just read gets its
261: // hopper selected, and moves to the Stacking Station.
262: // The card in the Check Station goes to the Reader Station, and we fill the
263: // Check Station with the next card (NULL if none).
264:
```

✓  
✓

```
265: void TCardReader::TransportCard(int hopper) {
266:
267:     if(StackStation != NULL) {
268:         StackStation -> Stack();
269:         delete StackStation;
270:     }
271:
272:     if(ReadStation != NULL) {
273:         ReadStation -> SelectStacker(hopperfd[hopper]);
274:     }
275:     ← StackStation = NULL;
276:     FI1402 -> SetReaderCheck(false);
277:     FI1402 -> SetReaderValidity(false);
278:     BusyEntry -> SetBusy(2);
279:     StackStation = ReadStation;
280:     ReadStation = CheckStation;
281:
282:     if(ready) {
283:         CheckStation = FeedCard();
284:     }
285:     else {
286:         CheckStation = NULL;
287:     }
288: }
289:
290: // Method to feed a card from the input hopper. Returns NULL if there is
291: // an EOF or I/O Error
292:
293: TCard *TCardReader::FeedCard() {
294:
295:     TCard *card;
296:     char temp[82];
297:     char *cp;
298:     int i;
299:
300:     // If the file isn't open, or we cannot allocate a card, return NULL
301:
302:     if(fd == NULL || (card = new TCard()) == NULL) {
303:         return(NULL);
304:     }
305:
306:     // Read up to 80 columns, looking for newline. If we hit EOF first,
307:     // throw away the card. Replace the newline with a blank. We have
308:     // to read up to 82 columns to get the newline.
309:
310:     for(cp = temp; cp - temp < 82; ++cp) {
311:         if(fd -> Read(cp,1) != 1) {
312:             delete card;
313:             delete fd;
314:             fd = NULL;
315:             return(card = NULL);
316:         }
317:         if(*cp == '\n') {
318:             break;
319:         }
320:     }
321:
322:     if(*cp != '\n') {
323:         DEBUG("TCardReader::FeedCard: No newline found within 82 characters",0);
324:     }
325:
326:     // If the character before the newline was a carriage return, throw it
327:     // away as well.
328:
329:     if(cp > temp && *(cp-1) == '\r') {
330:         *(cp-1) = ' ';
```

Channel → Hopper[Hopper]

```
331:     }
332:
333:     // Change the newline and any trailing garbage to blank.
334:
335:     for(; cp - temp < 80; ++cp) {
336:         *cp = ' ';
337:     }
338:
339:     // Finally, transfer the card image to the card object.
340:
341:     for(i=0, cp=temp; i < 81, 80; ++i, ++cp) {
342:         card -> image[i] = *cp;
343:     }
344:     return(card);
345: }
346:
347: // Method to read one card column. Returns BCD character as an integer, so
348: // that it also has a way to return EOF/Error status
349:
350: int TCardReader::DoInputColumn() {
351:
352:     int ch;
353:
354:     // If nothing in the buffer, say so.
355:
356:     if(ReadBuffer == NULL) {
357:         readerstatus |= IOCHNOTTRANSFER;
358:         Channel -> ExtEndofTransfer = true;
359:         return(-1);
360:     }
361:
362:     // If column is invalid, write to log, and take the reader offline.
363:
364:     if(column < 1 || column > 82, B1) {
365:         DEBUG("TCardReader::DoInputColumn: Invalid column: %d",column);
366:         ready = false;
367:         readerstatus |= IOCHNOTREADY;
368:         FI1402 -> SetReaderReady(false);
369:         Channel -> ExtEndofTransfer = true;
370:         return(-1);
371:     }
372:
373:     // If we have read all of the card, set end of transfer flag
374:
375:     if(column == 81) {
376:         ++column;                                // If it tries again, bad dog.
377:         Channel -> ExtEndofTransfer = true;
378:         return(-1);
379:     }
380:
381:     // Otherwise, grab a character from the card image. Check that it is
382:     // a valid character, and if not, signal a data check, and set the
383:     // reader check light.
384:
385:     ch = ReadBuffer -> image[column-1];
386:     ++column;
387:     if(BCD::BCDCheck(ch) < 0) {
388:         readerstatus |= IOCHDATACHECK;
389:         ready = false;
390:         FI1402 -> SetReaderCheck(true);
391:         FI1402 -> SetReaderValidity(true);
392:     }
393:     ch = BCD::BCDConvert(ch);                  // Invalid turns to alt b
394:     return(ch);
395: }
396:
```

*assert (column > 0 & column < 81);*

```
397: // Interface Methods for User Interface
398:
399: // Method to load the card file
400:
401: bool TCardReader::LoadFile(char *s) {
402:
403:     if(fd != NULL) {
404:         delete fd;
405:         fd = NULL;
406:     }
407:
408:     if(s == NULL) {
409:         return(false);
410:     }
411:
412:     try {
413:         fd = new TFileStream(s,fmOpenRead);
414:     }
415:     catch(EFOpenError &e) {
416:         return(false);
417:     }
418:
419:     strncpy(filename,s,MAXPATH);
420:     return(true);
421: }
422:
423: // Forced file close. Necessary because (apparently) the file open
424: // dialog checks and notices that the file is already open. This prevents
425: // reloading the same card deck twice in a row if we didn't have this method.
426:
427: // It also stacks all of the remaining cards in hopper 0.
428:
429: void TCardReader::CloseFile() {
430:
431:     if(fd != NULL) {
432:         delete fd;
433:         fd = NULL;
434:     }
435:
436:     while(StackStation != NULL) {
437:         StackStation -> SelectStacker(hopperfd[0]);
438:         StackStation -> Stack();
439:         TransportCard(0);
440:     }
441: }
442:
443: // Method to handle the STOP button
444:
445: void TCardReader::DoStop() {
446:     ready = false;
447:     eof = false;
448: }
449:
450: // Method to handle the START button
451:
452: bool TCardReader::DoStart() {
453:
454:     ready = false;
455:
456:     // Feed a card into the Check station if it is empty.
457:
458:     if(CheckStation == NULL) {
459:         CheckStation = FeedCard();
460:     }
461:
462:     // If there is no card at the Read Station, and there is (now) a card
```

```
463: // at the Check Station, move it to the Read Station, and
464: // feed another card.
465:
466: if(ReadStation == NULL && (ReadStation = CheckStation) != NULL) {
467:     CheckStation = FeedCard();
468: }
469:
470: // If there is a card at the Check Station, or, if EOF is pressed,
471: // there is at least a card at the Read Station, all is OK.
472:
473: if(CheckStation != NULL || (eof && ReadStation != NULL)) {
474:     ready = true;
475: }
476:
477: return(ready);
478: }
479:
```

```
class THopper : public TObject {
```

```
private:
```

```
TFFileStream * fd;
```

```
int Count;
```

```
char filename [MAXPATH];
```

```
public:
```

✓ bool Stack (TCard \*card); ✓

✓ inline void resetCount () { Count = 0; } ✓

✓ inline void incCount () { ++Count; } ✓

✓ bool setFile (char \*s); ✓

✓ THopper (); ✓

✓ inline int getCount () { return Count; } ✓

✓ inline char \*getFilename () { return filename; } ✓

```
TTHopper : T Hopper (TFTHopper *h) {
```

```
Count = 0;
```

```
fd = NULL;
```

```
Filename[8] = '10';
```

```
}
```

```
bool Stack (TCard *card) {
```

```
char *cp;
```

```
char temp [8];
```

```
incCount();  
if (fd == NULL) {  
    return (true);  
}
```

```
→ memcpy (temp, card->image, 8*4);
```

```
for (cp = temp+79; cp >= temp + cp == ' ' ; --cp) {
```

```
}
```

```
*++cp = 'r';
```

```
*++cp = 'n';
```

```
*++cp = '0';
```

```
try {
```

{ Write Error

```
bool THopper::setFile (char *s) {
    resetCount();
    if (fd != NULL) {
        delete fd;
        fd = NULL;
    }
    if (s == NULL) {
        return (false);
    }
    try {
        fd = new TFileStream(s, fmOpenCreate);
    }
    catch (EFOpenError &e) {
        return (false);
    }
    strcpy (filename, s, MAXPATH);
    return (true);
}
```

```
1: //-----
2: #ifndef UIREADERH
3: #define UIREADERH
4: //-----
5:
6: #define READER_IO_DEVICE 1
7:
8: // Very simple class (almost a "struct") to represent a card image
9:
10: class TCard : public TObject {
11:
12: public:
13:
14:     char image[80];                                // 80 columns of data
15:     bool Stack(TFileStream *fd);                  // Stack card to hopper
16: };
17:
18: // Class to implement the card reader interface and buffer
19:
20: class TCardReader : public T1410IODevice {
21:
22: protected:
23:
24:     TCard *ReadStation;                         // Station to read the card
25:     TCard *CheckStation;           StackStation // Where feed goes to
26:     TCard *ReadBuffer;                      // Where the read card goes
27:
28:     char filename[MAXPATH];                   // Input hopper, if you will
29:     TFileStream *fd;                        // Input hopper file stream
30:
31:     TFileStream *hopperfd[3];                // The 3 card read hoppers
32:
33:     int column;                           // Current column. 1-80, 81
34:
35:     bool ready;                          // True if reader ready
36:     bool eof;                            // True if EOF switch on
37:     bool buffertransferred;             // True if no data at Check Stn
38:
39:     int unit;                            // Hopper number: 0, 1, 2
40:
41:     int readerstatus;                  // Channel status value
42:     TBusyDevice *BusyEntry;
43:
44: public:
45:     // Implement the I/O device interface standard
46:     TCardReader(int devicenum, T1410Channel *Channel);
47:
48:     virtual int Select();
49:     virtual void DoOutput();
50:     virtual void DoInput();
51:     virtual int StatusSample();
52:     virtual void DoUnitControl(BCD opmod);
53:
54:     // State and status methods
55:
56:     inline bool IsReady() { return ready; }
57:     inline void SetEOF() { eof = true; }
58:     inline void ResetEOF() { eof = false; }
59:     inline bool IsBusy() { return BusyEntry -> TestBusy(); }
60:     inline char *GetFileName() { return filename; }
61:     inline bool SetUnit(int u);               // Check unit validity, set
62:     inline int GetUnit() { return unit; }
63:
64:     bool EOFLastCard();                    // True if EOF & last card read
65:     bool LoadFile(char *s);                // Call to open input file
66:
```

## UIREADER.h

```
67:     bool DoStart();                                // Process Start Button
68:     void DoStop();                                 // Process Stop Button
69:
70: private:
71:
72:     void TransportCard(int hopper);                // Transport card to hopper
73:     TCard *FeedCard();                             // Feed card from input file
74:     int DoInputColumn();                          // Process one card column
75:
76: };
77:
78:
79:
80: #endif
```

```
1: //-----
2: #include <vcl.h>
3: #pragma hdrstop
4:
5: #include <dir.h>
6: #include <stdio.h>
7: #include "UBCD.H"
8: #include "UI1410CPUUT.H"
9: #include "UI1410INST.H"
10: #include "UI1410CHANNEL.H"
11: #include "UIREADER.h"
12:
13: //-----
14: #pragma package(smart_init)
15:
16: #include "UI1410DEBUG.H"
17: #include "UI1402.H"
18:
19: // TCard Class Implementation
20:
21: bool TCard::Stack(TFileStream *fd) {
22:
23:     // TODO
24:
25:     return(true);
26: }
27:
28: // TCardReader Class Implementation
29:
30: // Constructor. Sets up the card reader
31:
32: TCardReader::TCardReader(int devicenum, T1410Channel *Channel) :
33:     T1410IODevice(devicenum, Channel) {
34:
35:     int i;
36:
37:     BusyEntry = new TBusyDevice();
38:
39:     filename[0] = '\0';
40:     fd = NULL;
41:
42:     ReadStation = NULL;
43:     CheckStation = NULL;
44:     ReadBuffer = NULL;           StackStation = NULL;
45:
46:     for(i=0; i < 3; ++i) {
47:         hopperfd[i] = NULL;
48:     }
49:
50:     ready = eof = buffertransferred = false;
51:     readerstatus = column = 0;
52: }
53:
54: // IO Device Implementation
55:
56: // Select. Returns 0 if successful, non 0 channel status otherwise
57:
58: int TCardReader::Select() {
59:
60:     int op;
61:
62:     column = 1;                      // Reset column back to start
63:
64:     // Check that the card reader is ready, not busy, that we are doing
65:     // either a read or unit control operation. If not, return appropriate
66:     // status.
```

```
67:     if(!IsReady()) {
68:         return(readerstatus = IOCHNOTREADY);
69:     }
70:     if(IsBusy()) {
71:         return(readerstatus = IOCHBUSY);
72:     }
73:     if(Channel -> ChWrite -> State()) {
74:         return(readerstatus = IOCHNOTREADY);
75:     }
76: }
77:
78: // If this is a read operation (opcode M or L), and there is
79: // nothing in the read buffer, return IOCHCONDITION and reset
80: // the eof switch status. (This works even for overlap because
81: // Select is called before the overlap starts). If that is all OK,
82: // call SetUnit to fill the read buffer and start the card transport.
83: // (Card transport for SSF is handled in the DoUnitControl method).
84:
85: readerstatus = 0;
86: if((op = CPU -> Op_Reg -> Get().To6Bit()) == OP_IO_MOVE || op == OP_IO_LOAD) {
87:     if(EOFLastCard()) {
88:         eof = false;
89:         FI1402 -> ResetEOF();
90:         FI1402 -> SetReaderReady(false);
91:         return(readerstatus = IOCHCONDITION);
92:     }
93:     if(!SetUnit(Channel -> GetUnitNumber())) {
94:         return(readerstatus = IOCHNOTREADY);
95:     }
96: }
97:
98: // Otherwise, AOK. (Might be IOCHCONDITION from SetUnit()).
99:
100: return(readerstatus);
101: }
102:
103: // Card reader unit control - used for Select Stacker and Feed instruction
104:
105: void TCardReader::DoUnitControl(BCD opmod) {
106:
107:     int hopper;
108:
109:     hopper = opmod.To6Bit();
110:
111:     switch(hopper) {
112:
113:     case 0:
114:     case 1:
115:     case 2:
116:         TransportCard(hopper);                                // Move cards along. Stack.
117:         buffertransferred = false;                            // We now have data avail.
118:         Channel -> UnitControlOverlapBusy = BusyEntry;    // Be busy for a while
119:         break;
120:
121:     default:
122:         DEBUG("TCardReader::DoUnitControl invalid unit: %d",hopper);
123:         Channel -> UnitControlOverlapBusy = NULL;
124:         readerstatus |= IOCHNOTREADY;
125:         Channel -> ExtEndoffTransfer = true;
126:         break;
127:     }
128:
129:     return;
130: }
131:
132: // Card Readers don't do ouptut very well, do they....
```

```
133:  
134: void TCardReader::DoOutput() {  
135:     readerstatus |= IOCHNOTREADY;  
136: }  
137:  
138: // Status sample is pretty simple for this device...  
139:  
140: int TCardReader::StatusSample() {  
141:     return(Channel -> GetStatus() | readerstatus);  
142: }  
143:  
144: // The real meat - input!  
145:  
146: void TCardReader::DoInput() {  
147:  
148:     bool wm = false;                                // True if wm in progress  
149:     BCD c;                                         // Column as BCD char  
150:     int card_input_char;                           // Column as BCD char too  
151:  
152:     // If the read buffer is empty, no transfer.  
153:     // (really, this should never happen!)  
154:  
155:     if(ReadBuffer == NULL) {  
156:         readerstatus |= IOCHNOTTRANSFER;  
157:         Channel -> ExtEndofTransfer = true;  
158:         return;  
159:     }  
160:  
161:     // Read the next column.  If return value is < 0, then something unusual  
162:     // happened.  channel status should already be set, so just return.  
163:  
164:     card_input_char = DoInputColumn();                // Get next column of data  
165:     if(card_input_char < 0) {  
166:         return;  
167:     }  
168:  
169:     // Handle load mode  
170:  
171:     if(Channel -> LoadMode && (card_input_char & 0x3f) == (BCD_WS & 0x3f)) {  
172:         wm = true;  
173:         card_input_char = DoInputColumn();              // Read char after ws  
174:         if(card_input_char < 0) {                      // Ooops -- off the end  
175:             return;  
176:         }  
177:         if((card_input_char & 0x3f) == (BCD_WS & 0x3f)) { // Another WS?  
178:             wm = false;                                // Yes. Throw away WM  
179:         }  
180:     }  
181:  
182:     // Convert the character to BCD.  The DoInputColumn routine already  
183:     // sets IOCHDATACHECK for invalid characters...  
184:  
185:     c = BCD(card_input_char);  
186:     c.SetOddParity();  
187:     if(wm) {  
188:         c.SetWM();  
189:         c.ComplementCheck();  
190:     }  
191:  
192:     // Tell the channel we have something for it!  
193:  
194:     Channel -> ChannelStrobe(c);  
195: }  
196:  
197: // Method to handle setting the unit during the select process for Read  
198:
```

## UIREADER.cpp

```

199: bool TCardReader::SetUnit(int u) {
200:
201:     switch(u) {
202:
203:     case 0:
204:     case 1:
205:     case 2:
206:         ReadBuffer = ReadStation;
207:         TransportCard(u);
208:         return(true);
209:
210:     case 9:
211:         if(buffertransferred) {
212:             readerstatus |= IOCHNOTTRANSFER;
213:             return(true);
214:         }
215:         ReadBuffer = ReadStation;
216:         return(true);
217:
218:     default:
219:         return(false);
220:     }
221: }
222:
223: // Method to transport the cards in the card reader. The card that was last
224: // read gets stacked (in the real reader this would be via the Stack station).
225: // The card in the Check Station goes to the Reader Station, and we fill the
226: // Check Station with the next card (NULL if none).
227:
228: void TCardReader::TransportCard(int hopper) {
229:
230:     if(ReadStation != NULL) {
231:         ReadStation -> Stack(hopperfd[hopper]); ←
232:         delete ReadStation; } SelectStacker() } { if(StackStation != NULL) {
233:     } } StackStation → Stack();
234:
235: FI1402 -> SetReaderCheck(false);
236: FI1402 -> SetReaderValidity(false);
237: BusyEntry -> SetBusy(2);
238: ReadStation = CheckStation; } { Delete StackStation;
239: CheckStation = FeedCard(); } } StackStation = ReadStation;
240: }
241:
242: // Method to feed a card from the input hopper. Returns NULL if there is
243: // an EOF or I/O Error
244:
245: TCard *TCardReader::FeedCard() {
246:
247:     TCard *card;
248:     char temp[81];
249:     char *cp; } 82
250:     int i;
251:
252:     // If the file isn't open, or we cannot allocate a card, return NULL
253:
254:     if(fd == NULL || (card = new TCard()) == NULL) {
255:         return(NULL);
256:     }
257:
258:     // Read up to 80 columns, looking for newline. If we hit EOF first,
259:     // throw away the card. Replace the newline with a blank. We have
260:     // to read up to 82 columns to get the newline.
261:
262:     for(cp = temp; cp - temp < 82; ++cp) {
263:         if(fd -> Read(cp,1) != 1) {
264:             delete card;
}

```

*✓*

*✓*

*✓*

```
265:             return(card = NULL);
266:         }
267:         if(*cp == '\n') {
268:             break;
269:         }
270:     }
271:
272:     if(*cp != '\n') {
273:         DEBUG("TCardReader::FeedCard: No newline found within 82 characters",0);
274:     }
275:
276: // If the character before the newline was a carriage return, throw it
277: // away as well.
278:
279:     if(cp > temp && *(cp-1) == '\r') {
280:         *(cp-1) = ' ';
281:     }
282:
283: // Change the newline and any trailing garbage to blank.
284:
285:     for(; cp - temp < 80; ++cp) {
286:         *cp = ' ';
287:     }
288:
289: // Finally, transfer the card image to the card object.
290:
291:     for(i=0, cp=temp; i < 81; ++i, ++cp) {
292:         card -> image[i] = *cp;
293:     }
294:     return(card);
295: }
296:
297: // Method to test for EOF button pressed and no more cards.
298:
299: bool TCardReader::EOFLastCard() {
300:     return(ReadStation == NULL && eof);
301: }
302:
303: // Method to read one card column. Returns BCD character as an integer, so
304: // that it also has a way to return EOF/Error status
305:
306: int TCardReader::DoInputColumn() {
307:
308:     int ch;
309:
310:     // If nothing in the buffer, say so.
311:
312:     if(ReadBuffer == NULL) {
313:         readerstatus |= IOCHNOTTRANSFER;
314:         Channel -> ExtEndofTransfer = true;
315:         return(-1);
316:     }
317:
318:     // If column is invalid, write to log, and take the reader offline.
319:
320:     if(column < 1 || column > 82) {
321:         DEBUG("TCardReader::DoInputColumn: Invalid column: %d",column);
322:         ready = false;
323:         readerstatus |= IOCHNOTREADY;
324:         FI1402 -> SetReaderReady(false);
325:         Channel -> ExtEndofTransfer = true;
326:         return(-1);
327:     }
328:
329:     // If we have read all of the card, set end of transfer flag
330:
```

```
331:     if(column == 81) {
332:         ++column;                                     // If it tries again, bad dog.
333:         Channel -> ExtEndofTransfer = true;
334:         return(-1);
335:     }
336:
337:     // Otherwise, grab a character from the card image. Check that it is
338:     // a valid character, and if not, signal a data check, and set the
339:     // reader check light.
340:
341:     ch = ReadBuffer -> image[column-1];
342:     ++column;
343:     if(BCD::BCDCheck(ch) < 0) {
344:         readerstatus |= IOCHDATACHECK;
345:         ready = false;
346:         FI1402 -> SetReaderCheck(true);
347:         FI1402 -> SetReaderValidity(true);
348:     }
349:     ch = BCD::BCDConvert(ch);                      // Invalid turns to alt b
350:     return(ch);
351: }
352:
353: // Interface Methods for User Interface
354:
355: // Method to load the card file
356:
357: bool TCardReader::LoadFile(char *s) {
358:
359:     if(fd != NULL) {
360:         delete fd;
361:         fd = NULL;
362:     }
363:
364:     if(s == NULL) {
365:         return(false);
366:     }
367:
368:     try {
369:         fd = new TFileStream(s,fmOpenRead);
370:     }
371:     catch(EFOpenError &e) {
372:         return(false);
373:     }
374:
375:     strncpy(filename,s,MAXPATH);
376:     return(true);
377: }
378:
379: // Method to handle the STOP button
380:
381: void TCardReader::DoStop() {
382:     ready = false;
383: }
384:
385: // Method to handle the START button
386:
387: bool TCardReader::DoStart() {
388:
389:     ready = false;
390:
391:     // If no cards, then we can't do anything.
392:
393:     if(fd == NULL) {
394:         return(false);
395:     }
396:
```

```
397: // Otherwise, feed a card into the Check station
398: CheckStation = FeedCard();
400:
401: // If there is a card there, move it to the Read Station, and
402: // feed another card.
403:
404: if((ReadStation = CheckStation) != NULL) {
405:     FeedCard();    CheckStation = FeedCard()
406: }
407:
408: // If there was at least one card, then everything is ready.
409:
410: if(ReadStation != NULL) {
411:     ready = true;
412: }
413:
414: return(ready);
415: }
416:
```

```
1: //-----
2: #ifndef UI1402H
3: #define UI1402H
4: //-----
5: #include <Classes.hpp>
6: #include <Controls.hpp>
7: #include <StdCtrls.hpp>
8: #include <Forms.hpp>
9: #include <Buttons.hpp>
10: #include <Dialogs.hpp>
11: //-----
12: class TFI1402 : public TForm
13: {
14:     __published:    // IDE-managed Components
15:     TBitBtn *ReaderStart;
16:     TBitBtn *ReaderStop;
17:     TBitBtn *EOF;
18:     TBitBtn *PunchStart;
19:     TBitBtn *PunchStop;
20:     TLabel *LightPunchReady;
21:     TLabel *LightPunchCheck;
22:     TLabel *LightPunchStop;
23:     TLabel *LightChips;
24:     TLabel *LightValidity;
25:     TLabel *LightFuse;
26:     TLabel *LightPower;
27:     TLabel *LightTransport;
28:     TLabel *LightStacker;
29:     TLabel *LightReaderReady;
30:     TLabel *LightReaderCheck;
31:     TLabel *LightReaderStop;
32:     TButton *LoadReaderHopper;
33:     TButton *LoadPunchHopper;
34:     TOpenDialog *FileOpenDialog;
35: private:      // User declarations
36: public:        // User declarations
37:     __fastcall TFI1402(TComponent* Owner);
38:
39:     void ResetEOF();
40:     void SetReaderCheck(bool flag);
41:     void SetReaderReady(bool flag);
42:     void SetPunchReady(bool flag);
43: };
44: //-----
45: extern PACKAGE TFI1402 *FI1402;
46: //-----
47: #endif
48:
```

```
1: //-----
2: #include <vcl.h>
3: #pragma hdrstop
4:
5: #include "UI1402.h"
6: //-----
7: #pragma package(smart_init)
8: #pragma resource "*.dfm"
9: TFI1402 *FI1402;
10: //-----
11: __fastcall TFI1402::TFI1402(TComponent* Owner)
12:   : TForm(Owner)
13: {
14:   Width = 632;
15:   Left = 260;
16:   Top = 300;
17:   Height = 164;
18:   WindowState = wsMinimized;
19: }
20: //-----
21:
22: // Method to reset card reader EOF status. In a real 1402, this would
23: // turn out the 1402 light (and enable the button). Here, we just enable
24: // the button.
25:
26: void TFI1402::ResetEOF() {
27:   EOF -> Enabled = true;
28: }
29:
30: // Method to set or reset the Reader Check light
31:
32: void TFI1402::SetReaderCheck(bool flag) {
33:   LightReaderCheck -> Enabled = flag;
34: }
35:
36: // Method to set or reset the Reader Ready Light
37:
38: void TFI1402::SetReaderReady(bool flag) {
39:   LightReaderReady -> Enabled = flag;
40:   ReaderStart -> Enabled = !flag;
41:   ReaderStop -> Enabled = flag;
42: }
43:
44: // And another to set or reset the Punch Ready light
45:
46: void TFI1402::SetPunchReady(bool flag) {
47:   LightPunchReady -> Enabled = flag;
48:   PunchStart -> Enabled = !flag;
49:   PunchStop -> Enabled = flag;
50: }
51:
52:
```

```
class TCard : public TObject {  
    int stacker;  
    char image[80];  
  
    void SelectStacker(int s), { stacker = s; }.  
    int getStacker () { return(stacker); }  
    bool Stack(TFileStream *fd);  
    bool SetImage(char *s);
```

};

class TCardReader : T1410Device {

protected:

char filename[MAXPATH];  
TFileStream \*fd;

TCard \*ReadStation;

TCard \*CheckStation;

TFileStream \*HopperFd[3];

int column;

bool ready;

bool eof;

TBusyDevice \*BusyEntry;

int unit;

int readerstatus;

BCD ch-char;

TCard \*ReadBuffer;

bool buffertransferred;

bool cardstacked; ?

- ✓ inline bool IsReady() { return ready; }
- ✓ inline bool EOFLastCard();
- ✓ inline void SetEOF() { eof = true; }
- ✓ inline char \*GetFileName() { return filename; }
- ✓ char \*LoadFile\_(char \*s);
- ✓ inline void ResetEOF() { eof = false; }
- ✓ inline bool IsBusy { return BusyEntry → TestBusy(); }

virtual int Select();

virtual void DoOutput();

virtual void DoInput();

virtual int StatusSample();

virtual void DoUnitControl(BCD opmod);

✓ bool TransportCard(int hopper);  
✓ bool SetUnit(int u);  
✓ inline int GetUnit() { return unit; }

TCard \* FeedCard()

✓ int DoInputColumn()

```
TCard Reader:: TCard Reader( int devicenum, TI410 Channel *channel) :  
    TI410 IO Device( devicenum, Channel) {  
  
    int i;  
    Filename [8] = '10';  
    fd = NULL;  
    SelectStation = NULL;  
    ReadStation = NULL;  
    CheckStation = NULL;  
    Read Buffer = NULL;  
    for (i=0; i<4; ++i) {  
        hopperfd [i] = NULL;  
    }  
  
    ready = eof = false;  
    BusyEntry = new TBusy Device();  
    readerstatus = 0;  
    column = 0;  
    unit = 0;  
    buffertransferred = false;  
    cardstacked = false;  
}
```

```
int TCordReader::Select() {
    column = 1;
    if (!IsReady())
        return (readerstatus = IOCHNOTREADY);
    }

    if (IsBusy())
        return (readerstatus = IOCHBUSY);

    if (column == 0)
        if (Channel->ChWrite->State())
            return (readerstatus = IOCHNOTREADY);
    }

    if (!SetUnit(Channel->GetUnitNumber()))
        return (readerstatus = IOCHNOTREADY);
    }

    return (readerstatus = 0);
}
```

```
void TCard Reader::DoUnitControl (BCD opmod) {
```

```
    int hopper;
```

```
    hopper = opmod.TOB.B.i();
```

```
    switch (hopper) {
```

```
        Case 0:
```

```
        Case 1:
```

```
        Case 2:
```

```
            TransportCard (hopper);
```

```
            buffertransferred = false;
```

```
            Channel → UnitControlOverlapBusy = BusyEntry;
```

```
            break;
```

```
        default:
```

```
            Channel → UnitControlOverlapBusy = NULL;
```

```
            readerstatus |= IOCHNOTREADY;
```

```
            Channel → ExtEndOfTransfer = true;
```

```
            break;
```

```
}
```

```
    return;
```

```
}
```

```
void TCardReader::DoOutput() {  
    readerstatus |= IOCHNOTREADY;  
    return;  
}
```

```
int TCardReader::StatusSample() {  
    return(Channel) → GetStatus() | readerstatus;  
}
```

①

```

void TCard Reader :: Do Input () {
    bool wM=false; BCD c; int card_input_char;
    if (Read Buffer == NULL) {
        reader status l = IOCHANOTTRANSFER;
        Channel → Extend of Transfer = true;
        return;
    }
    if (column < 1 || column > 81) {
        DEBUG ("TCard Reader :: Do Input Column Error %d", column);
        reader status l = IOCHANOTREADY;
        Channel → Extend of Transfer = true;
        return;
    }
    card_input_char = Do Input Column();
    if (card_input_char < 0)
        return;
}

```

;

~~if (card\_input\_char & 0x3f) == (BCD WS & 0x3f)~~

~~if ((card\_input\_char & 0x3f) == (BCD WS & 0x3f)) {~~

~~wM = false;~~

~~}~~

~~}~~

~~if (card\_input\_char & 0x3f) == (BCD WS & 0x3f)) {~~

~~wM = true;~~

~~}~~

~~}~~

Do Input

(2)

C. = BCD(card\_input-char);  
C. SetOddParity();  
if (wm) {

c. SetWM();  
d. Complement Check();

}

Channel → Channel Stroke(c);

}

```
boot TCard Reader:: SetUnit( int u ) {
```

```
    switch (u) {
```

```
        case 0:
```

```
        case 1:
```

```
        case 2:
```

```
            ReadBuffer = ReadStation;
```

```
            TransportCard(u);
```

```
            return (true);
```

```
            break;
```

```
        case 9:
```

```
            if (buffer transferred) {
```

```
                readerstate |= IOCHANOTTRANSFER;
```

```
                return (true);
```

```
}
```

```
            ReadBuffer = ReadStation;
```

```
            return (true);
```

```
            break;
```

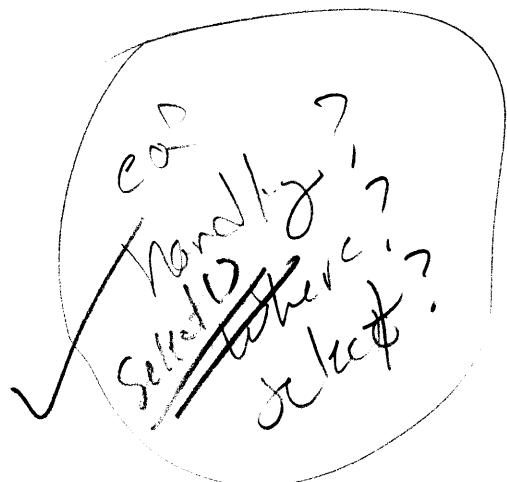
```
        default:
```

```
            return (false);
```

```
}
```

```
    return (false);
```

```
}
```



boo). TCardLoader :: TransportCard (int hopper) {

if (ReadStation != NULL) {

    ReadStation → SelectStacker (hopper);  
    ReadStation → Stack (hopperfd [hopper]);  
    delete ReadStation;

}

    BusyEntry → SetBusy (2);  
    ReadStation = CheckStation;  
    CheckStation = FeedCard ();  
    return (true);

}

TCard TCard Reader::FeedCard() {

    TCard \*card;  
    char \*cp;

    if (fd == NULL) {  
        return (NULL);

→     if ((card = new TCard()) == NULL)  
        return (NULL);  
    }

    for (cp = card->image; cp - card->image < 80; ++cp) {  
        if (fd->Read(cp, 1) != 1) {  
            delete card;  
            card = NULL;  
            break;  
        }  
        if (\*cp == '\n') {  
            break;  
        }

→     if (card != NULL) {  
        for (; cp - card->image < 80; ++cp) {  
            \*cp = ' ';

        if (cp > card->image + 80  
            &amp; (cp - 1) == '\r') {  
            \*(cp - 1) = '\n';

        }  
    }  
    return (card);

```
bool TCardReader::LoadFile (char *filename) {  
    if (fd != NULL) {  
        delete fd;  
        fd = NULL;  
    }  
    if (filename == NULL) {  
        return (false);  
    }  
    try {  
        fd = new TFileStream (filename, fmOpenRead);  
    }  
    catch (EFileError &e) {  
        return (false);  
    }  
    return (true);  
}
```

```
bool TCardReader::EOFLastCard() {
    return (ReadStation == NULL || eof);
}
```

```

int TCard Reader:: Do Input Column () {
    int ch;
    if (Read Buffer == NULL) {
        readerstatus = IOCHNOTREADY;
        Channel → Ext End of Transfer = true;
        return (-1);
    }
    if (column < 1 || column > 82) {
        DEBUG ("TCard Reader:: Do Input Column Invalid Column %d",
               column);
        readerstatus = IOCHNOTREADY;
        Channel → Ext End of Transfer = true;
        return (-1);
    }
    if (column == 81) {
        ++column;
        Channel → Ext End of Transfer = true;
        return (-1);
    }
    ch = Read Buffer → image [column - 1];
    ++column;
    if (BCD:: BCDCheck (ch) < 0) {
        readerstatus = IODATACHECK;
    }
    ch = BCD:: BCDConvert (ch);
    return (ch);
}

```

F1402

~~Reactor~~)

Reactor

Do Start()

Do Stop()

```
1: //-----
2: #include <vcl.h>
3: #pragma hdrstop
4:
5: #include <dir.h>
6: #include <stdio.h>
7: #include <assert.h>
8: #include "UBCD.H"
9: #include "UI1410CPUUT.H"
10: #include "UITAPEUNIT.h"
11:
12: //-----
13: #pragma package(smart_init)
14:
15: #include "UI1410DEBUG.H"
16:
17: #define TAPEDEBUG 1
18:
19: // Tape Unit Implementation.
20:
21: // Constructor
22:
23: TTapeUnit::TTapeUnit(int u) {
24:
25:     fd = NULL;
26:     BusyEntry = new TBusyDevice(); // Create a busy list entry
27:     Init(u); // Let common init take over
28: }
29:
30: // Initialization (not sure if anyone else will ever use this)
31:
32: void TTapeUnit::Init(int u) {
33:
34:     if(fd != NULL) {
35:         delete fd;
36:     }
37:
38:     unit = u;
39:     fd = NULL;
40:     loaded = fileprotect = tapeindicate = ready = selected = bot = false;
41:     write_irg = irg_read = modified = false;
42:     highdensity = true;
43:     filename[0] = '\0';
44:     record_number = 0;
45:     BusyEntry -> SetBusy(0); // Set not busy.
46:     return;
47: }
48:
49: // Method to reset file, if open, and reset flags. Typically called
50: // after an error of some sort.
51:
52: void TTapeUnit::ResetFile() {
53:
54:     if(fd != NULL) {
55:         delete fd;
56:     }
57:
58:     fd = NULL;
59:     ready = loaded = fileprotect = bot = false;
60:     irg_read = write_irg = modified = false;
61:     record_number = 0;
62:     return;
63: }
64:
65:
66: // Methods that interface to user interface buttons
```

```
67:
68: bool TTapeUnit::Reset() {
69:     ready = false;
70:     return(true);
71: }
72:
73: // Load the tape (file) (if not already loaded) and rewind.
74:
75: bool TTapeUnit::LoadRewind() {
76:
77:     if(ready) {                                     // Inop if drive is ready
78:         return(false);
79:     }
80:
81:     if(modified && write_irg) {                   // Write a closing 0 + IRG
82:         assert(fd != NULL);
83:         Write(0);
84:         modified = false;
85:     }
86:
87:     if(fd != NULL) {                                // If loaded, just rewind
88:
89:         try {
90:             fd -> Seek(0,soFromBeginning);
91:             irg_read = modified = false;
92:             write_irg = true;
93:             record_number = 0;
94:             return(bot = loaded = true);
95:         }
96:
97:         catch(char *dummy) {
98:             DEBUG("LoadRewind: Seek on failed on tape unit %d",unit);
99:             ResetFile();
100:            return(false);
101:        }
102:
103:    }
104:    else if(strlen(filename) == 0) {
105:        return(false);
106:    }
107:
108:    // Open the file. First try RW. If that fails, try RO and set fileprot.
109:
110:    try {
111:        fd = new TFileStream(filename,fmOpenReadWrite);
112:        fileprotect = false;
113:    }
114:
115:    catch(EFOpenError &e) {
116:
117:        try {
118:            fd = new TFileStream(filename,fmOpenRead);           // Read Only was OK
119:            fileprotect = true;
120:        }
121:
122:        catch(EFOpenError &e) {
123:            DEBUG("LoadRewind: open failed on tape unit %d",unit);
124:            DEBUG(e.Message.c_str(),0);
125:            ResetFile();
126:            return(false);
127:        }
128:    }
129:
130:    irg_read = modified = false;
131:    write_irg = true;
132:    record_number = 0;
```

```
133:     return(bot = loaded = true);
134: }
135:
136: // Unload the tape (file)
137:
138: bool TTapeUnit::Unload() {
139:
140:     if(ready || !loaded) {                                // If ready, ignore.
141:         return(false);
142:     }
143:
144:     ResetFile();                                         // Handles most of the work
145:     tapeindicate = false;
146:     return(true);
147: }
148:
149: // Mount a tape on the drive (associate a file)
150:
151: bool TTapeUnit::Mount(char *fname) {
152:
153:     if(ready || loaded) {                                // If ready or already
154:         return(false);                                  // loaded, ignore it.
155:     }
156:
157:     if(strlen(fname) == 0 || strlen(fname)+1 > sizeof(filename)) {
158:         return(false);
159:     }
160:     assert(fd == NULL);
161:     strcpy(filename,fname);
162:     irg_read = write_irg = fileprotect = tapeindicate = bot = modified = false;
163:
164:     return(true);
165: }
166:
167: // Start Button
168:
169: bool TTapeUnit::Start() {
170:
171:     if(ready || !loaded) {                                // If ready or not loaded
172:         return(false);                                  // can't help you!
173:     }
174:
175:     assert(fd != NULL);
176:
177:     return(ready = true);
178: }
179:
180: bool TTapeUnit::ChangeDensity() {
181:     if(ready) {
182:         return(false);
183:     }
184:     highdensity = !highdensity;
185:     return(true);
186: }
187:
188: // Methods that interface with the Tape Adapter Unit (TAU)
189:
190: bool TTapeUnit::Select(bool b) {
191:     selected = b;
192:     return(true);
193: }
194:
195: // Rewind to beginning of tape (file)
196:
197: bool TTapeUnit::Rewind() {
198:
```

```
199:     if(!selected || !loaded || !ready) {           // Must be ready to go...
200:         DEBUG("TTapeUnit::Rewind: Unit %d not selected or not ready",unit);
201:         return(false);
202:     }
203:
204: #ifdef TAPEDEBUG
205:     DEBUG("Rewind unit %d",unit);
206: #endif
207:
208:     assert(fd != NULL);
209:
210:     if(modified && write_irg) {                      // Mark end of record
211:         if(!Write(0)) {
212:             return(false);
213:         }
214:         write_irg = modified = false;
215:     }
216:
217:     try {
218:         fd -> Seek(0,soFromBeginning);
219:     }
220:
221:     catch(char *dummy) {
222:         DEBUG("Rewind: Seek failed, unit %d",unit);
223:         ResetFile();
224:         return(false);
225:     }
226:
227:     irg_read = modified = false;
228:     write_irg = true;
229:     BusyEntry -> SetBusy(record_number);           // Act like we are busy
230:     record_number = 0;
231:     return(bot = true);
232: }
233:
234: // Rewind and unload the tape (close the file)
235:
236: bool TTapeUnit::RewindUnload() {
237:
238:     if(!Rewind()) {                                // If rewind fails...
239:         ResetFile();
240:         return(false);
241:     }
242:
243:     ResetFile();
244:     tapeindicate = false;
245:     return(true);
246: }
247:
248: // Skip and blank tape, does nothing for now (until we have measured tape)
249:
250: bool TTapeUnit::Skip() {
251:
252: #ifdef TAPEDEBUG
253:     DEBUG("Write IRG (Skip) unit %d",unit);
254: #endif
255:
256:     if(!selected || !loaded || !ready) {
257:         return(false);
258:     }
259:     write_irg = true;
260:     return(true);
261: }
262:
263: // Space forward. (d-character is "A" - not in my Principles of Operation!)
264: // Basically pretty easy: all we have to do is call read until we get a
```

```
265: // negative return code, which will happen at EOF or IRG or an error.
266: // We can let the Tape Adapter Unit figure out the status.
267:
268: int TTapeUnit::Space() {
269:
270:     int rc;
271:
272: #ifdef TAPEDeBUG
273:     DEBUG("Space unit %d",unit);
274: #endif
275:
276:     while((rc = Read()) >= 0) {
277:         // Do nothing.
278:     }
279:     BusyEntry -> SetBusy(1); // Must go busy for a while
280:     return(rc);
281: }
282:
283: // Backspace. This one is a pain. To do it, we take two steps back, one
284: // forward. Repeatedly. Slow. Oh well....
285:
286: bool TTapeUnit::Backspace() {
287:
288:     if(!selected || !ready || !loaded) {
289:         DEBUG("TTapeUnit::Backspace: Unit %d not selected or not ready",unit);
290:         return(false);
291:     }
292:
293:     assert(fd != NULL);
294:
295: #ifdef TAPEDeBUG
296:     DEBUG("Backspace start: %d",fd -> Position);
297: #endif
298:
299:     if(bot) { // If at BOT, a NOP
300:         record_number = 0;
301:         return(true);
302:     }
303:
304:     BusyEntry -> SetBusy(1); // If not a BOT, go busy
305:
306:     // If we just ended a record, write out its IRG, then back up before it.
307:
308:     if(modified && write_irg) {
309:         if(!Write(0)) {
310:             ResetFile();
311:             return(false);
312:         }
313:         modified = write_irg = irg_read = false;
314:
315:         try {
316:             fd -> Seek(-1,soFromCurrent);
317:         }
318:
319:         catch(char *dummy) {
320:             DEBUG("Backspace: Seek over EOR failed on unit %d",unit);
321:             ResetFile();
322:             return(false);
323:         }
324:
325:     }
326:
327:     // Now, go into the two steps back, one step forward routine.
328:
329:     while(true) { // Start the dance...
330:
```

```
331:         // Seek back 2 characters
332:
333:         try {
334:             fd -> Seek(-2, soFromCurrent);
335:         }
336:
337:         catch(char *dummy) {
338:             DEBUG("Backspace: Seek failed on unit %d",unit);
339:             ResetFile();
340:             return(false);
341:         }
342:
343:         // If beginning of file, done! (special case)
344:
345:         if(fd -> Position == 0) {
346:             irg_read = false;
347:             bot = write_irg = true;
348:             record_number = 0;
349: #ifdef TAPEDEBUG
350:             DEBUG("Backspace end at BOT",0);
351: #endif
352:             return(true);
353:         }
354:
355:         // Read forward 1 character.  Quit on error or EOF
356:
357:         if(fd -> Read(&tape_buffer,1) != 1) {
358:             DEBUG("Backspace: Read failed: unexpected eof on unit %d",unit);
359:             return(false);
360:         }
361:
362:         // If we find an IRG bit on, we are done!  Have to leave the IRG
363:         // bit on the character, though.
364:
365:         if(tape_buffer & TAPE_IRG) {
366:             irg_read = write_irg = true;
367:             --record_number;
368: #ifdef TAPEDEBUG
369:             DEBUG("Backspace end: %d",fd -> Position);
370: #endif
371:             return(true);
372:         }
373:     }
374: }
375:
376: // Method to write a character.  Note that by this time any cute stuff
377: // (like parity, changing wordmarks in to word separators, changing
378: // word separators into two word separators, etc. should have already been
379: // handled in the TAU
380:
381: bool TTapeUnit::Write(char c) {
382:
383:     if(!loaded || !ready || TapeUnit !selected || fd == NULL) {
384:         DEBUG("TapeUnit::Write: Unit %d not ready or selected",unit);
385:         return(false);
386:     }
387:
388:     if(fileprotect) {
389:         DEBUG("TapeUnit::Write: Attempt to write when file protected, unit %d",
390:               unit);
391:         return(false);
392:     }
393:
394:     // If we have an irg left over from a previous read, back up over it.
395:
396:     if(irg_read) {
```

```
397:         try {
398:             DEBUG("Write seeking back over EOR from: %d",fd -> Position);
399:             fd -> Seek(-1,soFromCurrent);
400:             DEBUG("Write seeking back over EOR to: %d",fd -> Position);
401:             irg_read = false;
402:             write_irg = modified = true;           // Set modified to write IRG
403:         }
404:
405:         catch(char *dummy) {
406:             DEBUG("Write: Seek over EOR failed on unit %d",unit);
407:             ResetFile();
408:             return(false);
409:         }
410:     }
411:
412:     // If we have an IRG left to do from a previous write,
413:     // or, if the last operation was a read, set the IRG bit.
414:
415:     if(write_irg) {
416:         c |= TAPE_IRG;
417:     }
418:
419:
420: #ifdef TAPEDeBUG
421:     DEBUG("Write start: %d",fd -> Position);
422: #endif
423:
424:     if(fd -> Write(&c,1) != 1) {
425:         DEBUG("TapeUnit::Write: File I/O error writing on unit %d",unit);
426:         ResetFile();
427:         tapeindicate = true;
428:         return(false);
429:     }
430:
431:     irg_read = write_irg = false;
432:
433: #ifdef TAPEDeBUG
434:     DEBUG("Write end: %d",fd -> Position);
435: #endif
436:
437:     return(true);
438: }
439:
440: // Mark end of record. Called at the end of a transfer by the TAU.
441: // All this does is set the IRG flag for the start of the next record.
442:
443: void TTapeUnit::WriteIRG() {
444:     write_irg = modified = true;
445:     bot = irg_read = false;
446:     ++record_number;
447:
448: #ifdef TAPEDeBUG
449:     DEBUG("Write IRG unit %d",unit);
450: #endif
451:
452:     return;
453: }
454:
455: // Write tape mark. Just calls write to do the dirty work...
456: // Note that tape marks are *always* even parity. Since this writes
457: // an IRG and flushes, it also clears the modified flag.
458:
459: bool TTapeUnit::WriteTM() {
460:     bool status;
461:
462:     ++record_number;
```

```
463:
464: #ifdef TAPEDEBUG
465:     DEBUG("Write TM %d",unit);
466: #endif
467:
468:     if(!Write(TAPE_TM | TAPE_IRG)) {
469:         return(false);
470:     }
471:     irg_read = modified = bot = false;
472:     status = Write(TAPE_TM);
473:     write_irg = true;
474:     =modified
475: #ifdef TAPEDEBUG
476:     DEBUG("Write TM end: %d",fd -> Position);
477: #endif
478:
479:     return(status);
480: }
481:
482: // Read a character. Has to handle one interesting case. The tapes
483: // have an extra Tape Mark for each tape mark, so we have to skip that
484: // extra one. (The 2nd Tape Mark does NOT have IRG set).
485:
486: int TTapeUnit::Read() {
487:
488:     int rc;
489:
490:     if(!loaded || !ready || !selected || TapeUnit:: fd == NULL) {
491:         DEBUG("TapeUnit::Read: Unit not ready or selected: %d",unit);
492:         return(TAPEUNITNOTREADY);
493:     }
494:
495:     // Read a character, unless the last read resulted in an IRG, in
496:     // which case it is already in the buffer.
497:     // Interesting statuses are negative, just bubble them on up.
498:
499:     if(!irg_read) {
500:         if((rc = ReadNextChar()) < 0) {
501:             return(rc);
502:         }
503:         tape_buffer = (char) rc;
504:     }
505:
506:     // If at an irg and the character read is a TM, skip bogus next
507:     // char (or chars) until next IRG, and return EOF with Indicate.
508:     // A Tape Mark is only a Tape Mark as the first character. Also,
509:     // when we first see it at the end of a record, it just denotes the
510:     // IRG. (Which means we need to save it in tape_buffer, too)
511:
512:     if((bot || irg_read) &&
513:         (tape_buffer & 0x3f) == TAPE_TM &&
514:         (tape_buffer & TAPE_IRG) != 0) {
515: #ifdef TAPEDEBUG
516:         DEBUG("Tape Mark found, offset %d",fd -> Position);
517: #endif
518:         while((rc = ReadNextChar()) >= 0 && (rc & TAPE_IRG) == 0) {
519:             // Skip chars until next IRG.
520:         }
521:         ++record_number;
522:         bot = irg_read = false; // Assume physical EOF
523:         if(rc >= 0) {
524:             tape_buffer = (char) rc;
525:             irg_read = true; // Found char with IRG bit
526:         }
527:         tapeindicate = true;
528:         return(TAPEUNITEOF);
```

```
529:     }
530:
531:     // If we are at bot or an IRG, strip the IRG bit from the char.
532:     // We will then just return that char.
533:
534:     if(bot || irg_read) {
535:         tape_buffer &= (char) (~TAPE_IRG); —
536:         bot = irg_read = false;
537:     }
538:
539:     // If we hit the end of the record, then set irg now, and return such
540:
541:     if(tape_buffer & TAPE_IRG) {
542: #ifdef TAPEDEBUG
543:         DEBUG("TTapeUnit::Read: Found IRG character at %d", fd -> Position);
544: #endif
545:         irg_read = true;
546:         bot = false;
547:         ++record_number;
548:         return(TAPEUNITIRG);
549:     }
550:
551:     // Aw shucks, just return the blinkin' character already!
552:
553:     // Clear irg at this point too, because if we had an IRG bit,
554:     // it got handled in the preceeding IF.  If we hit EOF, the value
555:     // in the character is TAPEUNITEOF .
556:
557:     irg_read = false;
558:     return(tape_buffer);
559: }
560:
561: // Utility method to read a character from the file, and handle a few odds
562: // and ends.  Keeps us from having to do it all more than once...
563:
564: int TTapeUnit::ReadNextChar() {
565:
566:     unsigned char c;
567:
568: #ifdef TAPEDEBUG
569:     DEBUG("Tape Read start: %d", fd -> Position);
570: #endif
571:
572:     write_irg = true;                                // Next write must write IRG
573:
574:     if(fd -> Read(&c,1) != 1) {
575:         DEBUG("TapeUnit::ReadNextChar: Error or EOF in Read, unit %d", unit);
576:         tapeindicate = true;
577:         return(TAPEUNITEOF);
578:     }
579:     TAPE-TM | TAPE-IRG
580: #ifdef TAPEDEBUG
581:     DEBUG("Tape Read end: %d", fd -> Position);
582: #endif
583:     return(c);
584: }
585:
586:
```

```
1: //-----
2: #ifndef UI1403H
3: #define UI1403H
4: //-----
5: #include <Classes.hpp>
6: #include <Controls.hpp>
7: #include <StdCtrls.hpp>
8: #include <Forms.hpp>
9: #include <Buttons.hpp>
10: //-----
11:
12: #define PRINTPOSITIONS 132
13: #define PRINTMAXLINES 1500
14:
15: class TFI1403 : public TForm
16: {
17:     __published: // IDE-managed Components
18:     TMemo *Paper;
19:     TBitBtn *CheckReset;
20:     TBitBtn *Start;
21:     TBitBtn *Stop;
22:     TBitBtn *Space;
23:     TBitBtn *CarriageRestore;
24:     TBitBtn *SingleCycle;
25:     TLabel *LightPrintReady;
26:     TLabel *LightPrintCheck;
27:     TLabel *LightEndofForms;
28:     TBitBtn *CarriageStop;
29:     TLabel *LightFormsCheck;
30:     TLabel *LightSyncCheck;
31: private: // User declarations
32:
33:     int PrintPosition;
34:     char PrintBuffer[PRINTPOSITIONS + 1];
35:
36: public: // User declarations
37:     __fastcall TFI1403(TComponent* Owner);
38:     bool SendBCD(BCD c);
39:     bool DoPrint();
40:     bool NextLine();
41:
42:     TPrinter *PrinterIODevice;
43:
44: };
45: //-----
46: extern PACKAGE TFI1403 *FI1403;
47: //-----
48: #endif
```

Brennan ch 9

12

```
1: //-----
2: #include <vcl.h>
3: #pragma hdrstop
4:
5: #include <dir.h>
6:
7: #include "UBCD.H"
8: #include "UI1410CPUT.H"
9: #include "UI1410CHANNEL.H"
10: #include "UIPRINTER.h"
11: #include "UI1403.h"
12: //-----
13: #pragma package(smart_init)
14: #pragma resource "*.*dfm"
15: TFI1403 *FI1403;
16: //-----
17: __fastcall TFI1403::TFI1403(TComponent* Owner)
18:     : TForm(Owner)
19: {
20:     Width = 699;
21:     Left = 260;
22:     Top = 0;
23:     Height = 296;
24:     PrintPosition = 0;
25:
26:     WindowState = wsMinimized;
27: }
28: //-----
29:
30: bool TFI1403::SendBCD(BCD c) {
31:
32:     if(PrintPosition < 0 || PrintPosition >= PRINTPOSITIONS) {
33:         return(false);
34:     }
35:     PrintBuffer[PrintPosition++] = c.ToAscii();
36:     PrintBuffer[PrintPosition] = '\0';
37:     return(true);
38: }
39:
40: bool TFI1403::DoPrint() {
41:
42:     if(!PrintPosition) {
43:         return(true);
44:     }
45:     if(Paper -> Lines -> Capacity > PRINTMAXLINES) {
46:         Paper -> Lines -> Delete(0);
47:     }
48:     Paper -> Lines -> Add(PrintBuffer);
49:     PrintPosition = 0;
50:     return(true);
51: }
52:
53: bool TFI1403::NextLine() {
54:
55:     if(PrintPosition) {
56:         if(!DoPrint()) {
57:             return(false);
58:         }
59:     }
60:     strcpy(PrintBuffer, "\n");
61:     return(DoPrint());
62: }
63:
```

```
1: //-----
2: #ifndef UIPRINTERH
3: #define UIPRINTERH
4: //-----
5:
6: #define PRINTMAXFORM      1024
7: #define PRINTMAXTOKENS    80
8: #define PRINTCCMAXLINE   256
9:
10: #define PRINTER_IO_DEVICE   2
11:
12: //  Printer Adapter Unit (1414)
13:
14: class TPrinter : public T1410IODevice {
15:
16: protected:
17:
18:     int PrintStatus;                      // Printer Status for Channel
19:     bool Ready;                          // True if ready to go
20:     bool CarriageCheck;                 // True if runaway forms
21:     TBusyDevice *BusyEntry;             // Used to set delays
22:     int BufferPosition;                // Current output column
23:     int SkipLines;                     // Deferred Skip in lines
24:     int SkipChannel;                  // Deferred Skip to Channel
25:
26:     int FormLength;                   // Length of current form
27:     int FormLine;                     // Current line in form
28:     enum {
29:         FORM_SCREEN, FORM_FILE, FORM_PRINT
30:     } Forms;
31:
32:     TFileStream *ccfd;                // Carriage Control File
33:     int CarriageTape[PRINTMAXFORM];    // Carriage tape data
34:     char ccline[PRINTCCMAXLINE];
35:
36:     char FileName[MAXPATH];          // File name, if to file
37:     TFileStream *fd;                 // File FDs for print, CC file
38:
39: public:
40:
41:     TPrinter(int devicenum, T1410Channel *Channel);
42:
43:     inline bool IsReady() { return(Ready); }
44:     inline bool IsBusy() { return BusyEntry -> TestBusy(); }
45:
46:     void Start();                    // Called from UI Start Button
47:     void Stop();                     // Called from UI Stop Button
48:     void CheckReset();              // Called from UI Check Reset
49:
50:     bool CarriageRestore();         // Carriage to Channel 1
51:     void CarriageStop();           // Called from UI Carriage Stop
52:     bool CarriageSpace();          // Space to next line
53:
54:     virtual int Select();          // Channel Select
55:     virtual void DoOutput();       // Character to output
56:     virtual void DoInput();        // NOP on this device
57:     virtual int StatusSample();    // End of I/O status sample
58:     virtual void DoUnitControl(BCD opmod); // CC operation
59:
60:     void ControlCarriage(BCD opmod); // Send one char of output
61:     void DoOutputChar(BCD c);      // Flush line to device.
62:     void DoPrint();               // Output to a file or printer
63:     // TODO  bool PrintAscii(char c); // Output to a file or printer
64:
65:     int SetCarriageTape(char *tapefile); // Set up carriage control
66:     void SetCarriageDefault();      // Set default CC tape
```

```
67:     bool CarriageChannelTest(int cchannel);      // True if this line has chan.
68:
69: private:
70:
71:     bool CarriageSkip(int lines, int channel);
72:     bool GetCarriageLine();
73:     void ParseCarriageLine(char *line, char **elements);
74:     int CarriageTapeError(int rc);                  // Cleans up after tape errors
75:
76: };
77:
78: #endif
```

```
1: //-----
2: #include <vc1.h>
3: #pragma hdrstop
4:
5: #include <dir.h>
6: #include <assert.h>
7: #include <stdlib.h>
8: #include <ctype.h>
9: #include "UBCD.H"
10: #include "UI1410CPUH.H"
11: #include "UI1410CHANNEL.H"
12: #include "UIPRINTER.h"
13:
14: //-----
15: #pragma package(smart_init)
16:
17: #include "UI1403.H"
18:
19: // Printer Adapter Unit Implementation. Follows I/O Device Interface.
20:
21: // Constructor. Creates a printer! (NOTE: RIGHT NOW, CAN ONLY BE 1 !!)
22:
23: TPrinter::TPrinter(int devicenum, T1410Channel *Channel) :
24:     T1410IODevice(devicenum, Channel) {
25:
26:     BusyEntry = new TBusyDevice();
27:     Forms = FORM_SCREEN;                                // Default: Print to screen
28:
29:     ccfd = NULL;
30:     ccline[0] = '\0';
31:     SetCarriageDefault();                             // Default CC tape
32:
33:     FileName[0] = '\0';
34:     fd = NULL;
35:
36:     SkipLines = SkipChannel = 0;
37:     BufferPosition = 0;
38:
39:     PrintStatus = 0;
40:     Start();
41: }
42:
43: // Start. Called from initialization and when Start button on printer
44: // user interface is pressed.
45:
46: void TPrinter::Start() {
47:     if(CarriageCheck) {
48:         return;
49:     }
50:     Ready = true;
51:     FI1403 -> LightPrintReady -> Enabled = true;
52: }
53:
54: // Stop. Called when Stop button on printer user interface is pressed.
55:
56: void TPrinter::Stop() {
57:     Ready = false;
58:     FI1403 -> LightPrintReady -> Enabled = false;
59: }
60:
61: // CheckReset: You guessed it: Called when the Check Reset button on the
62: // printer user interface is pressed.
63:
64: void TPrinter::CheckReset() {
65:     CarriageCheck = false;
66:     FI1403 -> LightFormsCheck -> Enabled = false;
```

```
67: }
68:
69: // Carriage Restore: You know the routine.
70:
71: bool TPrinter::CarriageRestore() {
72:     return(CarriageSkip(0,1));
73: }
74:
75: // Carriage Space: Ditto
76:
77: bool TPrinter::CarriageSpace() {
78:     return(CarriageSkip(1,0));
79: }
80:
81: // Carriage Stop
82:
83: void TPrinter::CarriageStop() {
84:     Stop();
85:     CarriageCheck = true;
86:     FI1403 -> LightFormsCheck -> Enabled = true;
87: }
88:
89: // Now, for the I/O Device Interface.
90:
91: // Select is called at the beginning of an IO operation.
92:
93: int TPrinter::Select() {
94:
95:     if(!Ready) {
96:         return(IOCHNOTREADY);
97:     }
98:     if(BusyEntry -> TestBusy()) {
99:         return(IOCHBUSY);
100:    }
101:    BufferPosition = 0; /
102:    if(Channel -> ChRead -> State() ||
103:        (Channel -> GetUnitNumber() != 0 && Channel -> GetUnitNumber() != 1) ) {
104:        return(IOCHNOTREADY);
105:    }
106:    return(PrintStatus = 0);
107: }
108:
109: // Status Sample is called at the end of an I/O operation.
110:
111: int TPrinter::StatusSample() {
112:     return(Channel -> GetStatus() | PrintStatus); ←
113: }
114:
115: // Here is where the real work gets done...
116:
117: void TPrinter::DoOutput() {
118:
119:     BCD ch_char;
120:
121:     // If not ready, say so.
122:
123:     if(!Ready) {
124:         PrintStatus |= IOCHNOTREADY;
125:         Channel -> ExtEndofTransfer = true;
126:         return;
127:     }
128:
129:     // Get character from the channel silo
130:
131:     ch_char = Channel -> ChR2 -> Get(); ←
132:
```

if (BufferPosition !=  
PRINTPOSITIONS)  
PrintStatus |= IOCHNOTREADY;

} ←

```
133: // If we have too many characters, indicate the problem
134:
135: if(BufferPosition >= PRINTPOSITIONS) {
136:     PrintStatus |= IOCHWLRECORD;
137:     Channel -> ExtEndofTransfer = true;
138:     return;
139: }
140:
141: // Check the parity of the character coming from memory
142:
143: if(!ch_char.CheckParity()) {
144:     PrintStatus |= IOCHDATACHECK;
145: }
146:
147: // In Load Mode, we turn Wordmarks into Word Separators.
148: // The addition of the word separator makes a Wrong Length Record very
149: // very likely!
150:
151: if(Channel -> LoadMode && ch_char.TestWM()) {
152:     DoOutputChar(BCD_WS);
153:     if(++BufferPosition >= PRINTPOSITIONS) {
154:         PrintStatus |= IOCHWLRECORD;
155:         Channel -> ExtEndofTransfer;
156:         return;
157:     }
158: }
159:
160: // Having handled Load Mode, we proceed. For the normal print unit (%20),
161: // we now print the character. If we are in Move Mode and have the
162: // alternate print unit (%21), we print a space, because we have already
163: // stripped the WM. (Not sure this is how it really worked, but the
164: // Principles of Operation indicates that a L%21 prints a blank line).
165:
166: if(Channel -> GetUnitNumber() == 0) {
167:     DoOutputChar(ch_char);
168: }
169: else { // %21
170:     DoOutputChar(Channel -> MoveMode && ch_char.TestWM() ?
171:                     BCD_1 : BCD_SPACE);
172: }
173: ✓ ++BufferPosition;
174:
175: // Check for end of transfer. If we have need more characters,
176: // let the Channel know. Otherwise, let the Channel know we have
177: // had enough.
178:
179: // Also, this is normally how we will know we are ready to print the
180: // line. (A Wrong Length Record can suppress output, according to the
181: // Principles of Operation). It is easier to do it here than wait to
182: // do it a StatusSample time - which has to service both Print and
183: // Carriage Control operations. After we print the line, we might
184: // also have a deferred Carriage Control operation to do.
185:
186: if(BufferPosition < PRINTPOSITIONS) {
187:     Channel -> OutputRequest = true;
188:     Channel -> CycleRequired = true;
189: }
190: else {
191:     DoPrint();
192:     Channel -> ExtEndofTransfer = true; ✓
193:     if(SkipLines || SkipChannel) {
194:         CarriageSkip(SkipLines, SkipChannel);
195:         SkipLines = SkipChannel = 0;
196:     }
197:     else {
198:         CarriageSpace();
```

```
199:         }
200:         BusyEntry -> SetBusy(2);
201:     }
202: }
203:
204: // Send the output character to the appropriate device.
205:
206: void TPrinter::DoOutputChar(BCD c) {
207:
208:     bool status;
209:
210:     c = c & (BIT_NUM | BIT_ZONE);
211:
212:     switch(Forms) {
213:
214:     case FORM_SCREEN:
215:         status = FI1403 -> SendBCD(c);
216:         break;
217:
218:     // TODO: FILE and PRINT spool
219:
220:     default:
221:         assert(false);
222:     }
223:
224:     if(!status) {
225:         PrintStatus |= IOCHCONDITION;
226:     }
227: }
228:
229: // Send the word to the appropriate device to actually print the line.
230:
231: void TPrinter::DoPrint() {
232:
233:     bool status;
234:
235:     switch(Forms) {
236:
237:     case FORM_SCREEN:
238:         status = FI1403 -> DoPrint();
239:         break;
240:
241:     // TODO: File and Print Spool
242:
243:     default:
244:         assert(false);
245:     }
246:
247:     if(!status) {
248:         PrintStatus |= IOCHCONDITION;
249:     }
250: }
251:
252: // Carriage Control operations. Basically pretty simple stuff.
253:
254: void TPrinter::DoUnitControl(BCD opmod) {
255:
256:     Channel -> UnitControlOverlapBusy = NULL;
257:     if(!Ready) {
258:         PrintStatus |= IOCHNOTREADY;
259:     }
260:     else {
261:         ControlCarriage(opmod);
262:     }
263:     Channel -> ExtEndofTransfer = true;
264: }
```

```
265:  
266: void TPrinter::ControlCarriage(BCD opmod) {  
267:  
268:     int dint;  
269:     int num;  
270:  
271:     dint = opmod.To6Bit();  
272:     num = dint & BIT_NUM;  
273:  
274:     // Single spaces get handled by default (may change later)  
275:  
276:     if(num == 0) {  
277:         return;  
278:     }  
279:  
280:     SkipChannel = SkipLines = 0;  
281:  
282:     // 0 - 9, #, @: Skip to carriage tape channels 1 - 12  
283:  
284:     if((dint & BIT_ZONE) == 0) {  
285:         if(num > 12) {  
286:             return;  
287:         }  
288:         CarriageSkip(0,num);  
289:     }  
290:  
291:     // A -I, ? . <lozenge>: Skip to channels 1-2 *after* printing  
292:  
293:     else if((dint & BIT_ZONE) == BIT_ZONE) {  
294:         if(num <= 12) {  
295:             SkipChannel = num;  
296:         }  
297:     }  
298:  
299:     // J, K, L: Immediate space of 1, 2 or 3 lines  
300:  
301:     else if((dint & BIT_ZONE) == BITB) {  
302:         if(num > 3) {  
303:             return;  
304:         }  
305:         CarriageSkip(num,0);  
306:     }  
307:  
308:     // /, S, T: Skip 1, 2 or 3 lines *after* printing  
309:  
310:     else {  
311:         if(num > 3) {  
312:             return;  
313:         }  
314:         SkipLines = num;  
315:     }  
316: }  
317:  
318: // Printers ignore input. (Actually, this should never get called, because  
319: // select will return Not Ready on an attempt to select for input).  
320:  
321: void TPrinter::DoInput() {  
322:     PrintStatus |= IOCHNOTREADY;  
323:     return;  
324: }  
325:  
326: // PRIVATE routine to skip carriage to given channel or given number of lines.  
327:  
328: bool TPrinter::CarriageSkip(int spaces, int chan) {  
329:  
330:     int num;
```

```

331:     int line;                               φ
332:
333:     if(spaces < 0 || spaces > 4 || chan < 8 || chan > 12) {
334:         PI1403 -> LightFormsCheck -> Enabled = true;
335:         CarriageCheck = true;           ✓ CarriageStop()
336:         return(Ready = false);
337:     }
338:
339:     if(spaces > 0 && spaces < 4) {
340:         while(spaces--) {
341:             CarriageSpace();
342:         }
343:         return(true);
344:     }
345:
346:     num = (1 << (chan - 1));               } { if (chan == 0) {
347:
348:     for(line=0; line <= FormLength; ++line) {
349:         CarriageSpace();
350:         if(CarriageChannelTest(num)) {
351:             BusyEntry -> SetBusy(2);
352:             return(true);
353:         }
354:     }
355:
356:     PI1403 -> LightFormsCheck -> Enabled = true;
357:     CarriageCheck = true;           ✓ CarriageStop()
358:     return(Ready = false);
359: }
360:
361: bool TPrinter::CarriageChannelTest(int ch) {
362:     return((CarriageTape[FormLine] & ch) != 0);
363: }
364:
365: // The remaining code has to do with setting up the carriage control tape.
366: // A carriage control tape is read in from a file using the same format
367: // as the Newcomer 1401 simulator.
368:
369: // * beginning a line indicates a comment
370: // LENGTH ###
371: // CHAN # = # [[+]]# ...  

372:
373: // Method to set up a default carriage tape, during construction
374:
375: void TPrinter::SetCarriageDefault() {
376:
377:     int line;
378:
379:     for(line=0; line < PRINTMAXFORM; ++line) {
380:         CarriageTape[line] = 0;
381:     }
382:
383:     FormLength = 66;
384:     FormLine = 1;                         ↓
385:     CarriageTape[4] = 1 | 2 | 4 | 8 | 16 | 32 | 64 | 128 | 512 | 1024;
386:     // Channels      1   2   3   4   5   6   7   8   10   11
387:
388:     CarriageTape[61] = 256;    // Channel 9
389:     CarriageTape[63] = 2048;   // Channel 12
390: }
391:
392: // Main method for carriage tape - sets up carriage tape from a file
393: // Returns 0 if tape is OK. Returns - value if an error. The value
394: // is in fact the line number.
395:
396: int TPrinter::SetCarriageTape(char *filename) {

```

```
397:     char *elements[PRINTMAXTOKENS];           // Up to 80 fields per line.
398:     int line, chan, i, n;
399:
400:
401:     for(line=0; line < PRINTMAXFORM; ++line) {
402:         CarriageTape[line] = 0;
403:     }
404:
405:     // Passing us no file is OK: Means they want the default carriage tape.
406:
407:     if(filename == NULL || strlen(filename) == 0) {
408:         SetCarriageDefault();
409:         return(0);
410:     }
411:
412:     assert(ccfd == NULL);
413:     line = 0;
414:
415:     // Try and open the file. If it fails, report an error on line 1.
416:
417:     try {
418:         ccfд = new TFileStream(filename, fmOpenRead);
419:     }
420:     catch(EFOpenError &e) {
421:         return(CarriageTapeError(-1));
422:     }
423:
424:     // Skip any leading comment lines. Error if we hit EOF.
425:
426:     do {
427:         if(!GetCarriageLine()) {
428:             return(CarriageTapeError(-line));
429:         }
430:         ++line;
431:     } while(ccline[0] != '*');
432:
433:     // Parse the line. It should have 2 elements. LENGTH and the number
434:
435:     ParseCarriageLine(ccline,elements);
436:     if(strcmp(elements[0],"LENGTH") == 0) {
437:         return(CarriageTapeError(-line));
438:     }
439:     FormLength = atoi(elements[1]);
440:     if(FormLength < 1 || FormLength > PRINTMAXFORM) {
441:         return(CarriageTapeError(-line));
442:     }
443:
444:     // Now process the carriage control lines themselves.
445:
446:     n = 0;
447:     while(GetCarriageLine()) {
448:         ++line;
449:         if(ccline[0] == '*') {                      // * for comment
450:             continue;
451:         }
452:         ParseCarriageLine(ccline,elements);
453:         if(strcmp(elements[0],"CHAN") != 0 ||      // Check CHAN # =
454:             strcmp(elements[2],"=") != 0) {
455:             return(CarriageTapeError(-line));
456:         }
457:         chan = atoi(elements[1]);                  // Validate channel number
458:         if(chan < 1 || chan > 12) {
459:             return(CarriageTapeError(-line));
460:         }
461:         for(i=3; elements[i] != NULL; ++i) {        // Process the form lines
462:             if(*elements[i] == '+') {
```

```
463:             n += atoi(elements[i]+1);
464:         }
465:     else {
466:         n = atoi(elements[i]);
467:     }
468:     if(n < 1 || n > FormLength) {
469:         CarriageTapeError(-line);
470:     }
471:     CarriageTape[n] |= (1 << (chan - 1));
472: }
473: }
474:
475: FormLine = 1;
476: delete ccfd;
477: ccfд = NULL;
478: return(0);
479: }
480:
481: // Utility method to clean up after carriage tape errors
482:
483: int TPrinter::CarriageTapeError(int rc) {
484:
485:     if(ccfd != NULL) {
486:         delete ccfд;
487:     }
488:     ccfд = NULL;
489:     SetCarriageDefault();
490:     return(rc);
491: }
492:
493: // Method to get one line of input. (Why Borland didn't have this kind of
494: // method as part of their file stream object I have *no* idea!
495:
496: bool TPrinter::GetCarriageLine() {
497:
498:     char *cp;
499:
500:     if(ccfd == NULL) {
501:         return(false);
502:     }
503:
504:     for(cp = ccline; cp - ccline < PRINTCCMAXLINE; ++cp) {
505:         if(ccfd -> Read(cp,1) != 1) {
506:             return(false);
507:         }
508:         if(*cp == '\n') {
509:             *cp = 0;
510:             return(true);
511:         }
512:     }
513:
514:     ccline[PRINTCCMAXLINE] = 0;
515:     return(false);
516: }
517:
518: // Method to set pointers to the tokens in a line.
519:
520: void TPrinter::ParseCarriageLine(char *line, char **element) {
521:
522:     int i=0;
523:     element[0] = NULL;
524:
525:     // Ignore leading white space to find first token
526:
527:     while(*line && isspace(*line)) {
528:         ++line;
```

```
529:     }
530:
531: // Process tokens one at a time until end of line.
532:
533: while(*line && i < PRINTMAXTOKENS-1) {
534:     element[i] = line;
535:     element[i+1] = NULL;
536:     while(*++line && !isspace(*line)) {
537:         // Skip over rest of the token
538:     }
539:     if(!*line) {
540:         break;
541:     }
542:     *line = 0;                                // Terminate token with '\0'
543:     while(*++line && isspace(*line)) {
544:         // Skip over white space
545:     }
546: }
547: }
548:
```

- ① No work on short write
- ② Comm check on normal write (if conditions)
- ③ UI Buttons NOP (no code yet!)
- ④ Cleaning up Printer  $\leftrightarrow$  OR interface -  
use methods instead of direct access
- ⑤ Implement CC instructions.
- bug problem (Comp. Rent)  
(STOP POU)
- ⑥ Handover
- Manual operations only OK if STOPPED  $\rightarrow$  No work  
Auto operations only OK if Ready  $\leftarrow$  OK

```
1: //-----
2: #include <vcl.h>
3: #pragma hdrstop
4:
5: #include <dir.h>
6: #include <stdio.h>
7: #include <assert.h>
8: #include <errno.h>
9: #include "UBCD.H"
10: #include "UI1410CPUH.H"
11: #include "UI1410CHANNEL.H"
12: #include "UITAPEUNIT.H"
13: #include "UITAPETAU.H"
14: #include "UI729TAPE.H"
15:
16:
17: //-----
18: #pragma package(smart_init)
19:
20: #include "UI1410DEBUG.H"
21:
22: // 1410 Tape Adapter Unit Implementation. Follows I/O Device Interface.
23:
24: // Constructor. Creates tape drives!
25: // NOTE: This is expected to be called with the EVEN PARITY designation
26: // for the device number ("U"). This constructor automatically adds the
27: // ODD PARITY designation ("B")!
28:
29: TTapeTAU::TTapeTAU(int devicenum, T1410Channel *Channel) :
30:     T1410IODevice(devicenum, Channel) {
31:
32:     static char parity_table[128] = {
33:         0,1,1,0,1,0,0,1,1,0,0,1,0,1,0,0,1,1,0,0,1,1,0,0,1,0,0,1,
34:         1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,
35:         1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,
36:         0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,1,0,0,1,0,1,1,0
37:     };
38:
39:     int i;
40:
41:     for(i=0; i < 10; ++i) {
42:         Unit[i] = new TTapeUnit(i);
43:     }
44:
45:     TapeUnit = NULL;
46:     tapestatus = 0;
47:     chars_transferred = 0;
48:     tape_parity_table = parity_table;
49:
50:     // Create the odd parity device now...
51:
52:     Channel -> AddIODevice(this, BCD(BCD::BCDConvert('B')).To6Bit());
53: }
54:
55: // Select. If some other unit is selected, it gets deselected.
56: // The unit comes from the CPU / Channel
57:
58: int TTapeTAU::Select() {
59:
60:     int u;
61:
62:     if(TapeUnit != NULL) {
63:         TapeUnit -> Select(false);
64:         TapeUnit = NULL;
65:     }
66:
```

```
67:     // Find the tape unit. Give up if none there. Otherwise, select it.
68:
69:     u = Channel -> GetUnitNumber();
70:     if((TapeUnit = GetUnit(u)) == NULL) {
71:         return(tapestatus = IOCHNOTREADY);
72:     }
73:     TapeUnit -> Select(true);
74:
75:     // If the unit isn't ready, say so
76:
77:     if(!TapeUnit -> IsReady()) {
78:         return(tapestatus = IOCHNOTREADY);
79:     }
80:
81:     // If the unit is busy, say so
82:
83:     if(TapeUnit -> IsBusy()) {
84:         return(tapestatus = IOCHBUSY);
85:     }
86:
87:     // Set 0 length record so far.
88:
89:     chars_transferred = 0;
90:
91:     // If writing, and tape is write protected, set not ready and condition.
92:     // Not sure if that is write - the diagnostics will probably figure it out
93:     // for me... 8-
94:
95:     if(Channel -> ChWrite -> State() && TapeUnit -> IsFileProtected()) {
96:         return(tapestatus |= (IOCHNOTREADY | IOCHCONDITION));
97:     }
98:
99:     return(tapestatus = 0);
100: }
101:
102: // Return a pointer to a given tape drive, NULL if invalid.
103:
104: TTapeUnit *TTapeTAU::GetUnit(int u) {
105:
106:     if(u < -0 || u > 9) {
107:         return(NULL);
108:     }
109:     return(Unit[u]);
110: }
111:
112: // DoOutput: Accept an output character from the channel.
113:
114: void TTapeTAU::DoOutput() {
115:
116:     // If no (or invalid) unit selected, just return not ready.
117:
118:     if(TapeUnit == NULL) {
119:         tapesstatus |= IOCHNOTREADY;
120:         Channel -> ExtEndofTransfer = true;
121:         return;
122:     }
123:
124:     // If file protected, return not ready as well. (Not sure if this is
125:     // correct, but hopefully the diagnostics will test it)
126:
127:     if(TapeUnit -> IsFileProtected()) {
128:         tapesstatus |= IOCHNOTREADY;
129:         Channel -> ExtEndofTransfer = true;
130:         return;
131:     }
132:
```

```
133: // Get the character from the channel. Strip down to 6 bits for tape.
134:
135: ch_char = Channel -> Chr2 -> Get();
136: tape_char = ch_char.To6Bit();
137:
138: // If we are in load mode, and the character either has a word mark or is
139: // itself a word separator, write out a word separator character to tape.
140:
141: if((Channel -> LoadMode &&(ch_char.TestWM() || 
142: tape_char == (BCD_WS & 0x3f))) {
143:     if(!DoOutputWrite(BCD_WS)) {
144:         tapestatus |= IOCHCONDITION;
145:         Channel -> ExtEndofTransfer = true;
146:         return;
147:     }
148: }
149:
150: // If the incoming character has invalid parity, flag a datacheck. But
151: // we will still write what we can, in valid parity, to tape.
152:
153: if(!ch_char.CheckParity()) {
154:     tapestatus |= IOCHDATACHECK;
155: }
156:
157: // Finally, write out the character.
158:
159:
160: if(!DoOutputWrite(tape_char)) {
161:     tapestatus |= IOCHCONDITION;
162:     Channel -> ExtEndofTransfer = true;
163:     return;
164: }
165:
166: Channel -> OutputRequest = true;
167: Channel -> CycleRequired = true;
168:
169: ++chars_transferred;
170: }
171:
172: // Method to tell channel when we have data.
173:
174: void TTapeTAU::DoInput() {
175:
176:     bool wm = false;
177:     BCD b;
178:
179:     if(TapeUnit == NULL) {
180:         tapestatus |= IOCHNOTREADY;
181:         Channel -> ExtEndofTransfer = true;
182:         return;
183:     }
184:
185:     // Get a character from the tape unit. If an interesting status, just
186:     // return. The utility method will have already set the right status.
187:     // (That is why the utility method exists!)
188:
189:     tape_read_char = DoInputRead();
190:     if(tape_read_char < 0) {
191:         return;
192:     }
193:
194:     ++chars_transferred;
195:
196:     // If we got a tape mark, tell the channel to store it. The TAU always
197:     // returns tape marks in EVEN PARITY. The channel will store it as is
198:     // (and flag a machine check) unless asterisk insert is on!
```

```
199: if((tapestatus & IOCHCONDITION) &&
200:     tape_read_char == BCD_TM) {
201:     Channel->ChannelStrobe(BCD_TM);
202:     return;
203: }
204:
205:
206: // If we are reading in load mode, and we got a word separator, read the
207: // next character. Then we will either set a WM on it, or, if it too is
208: // a word separator, just store the word separator. If we have a problem
209: // reading the next char, just return -- the read routine will have set
210: // status -- that is why it is there! 8-
211:
212: if(Channel->LoadMode && (tape_read_char & 0x3f) == (BCD_WS & 0x3f)) {
213:     wm = true;
214:     tape_read_char = DoInputRead();
215:     if(tape_read_char < 0) {
216:         return;
217:     }
218:     if((tape_read_char & 0x3f) == BCD_WS) {
219:         wm = false;
220:     }
221: }
222:
223: b = BCD(tape_read_char);
224:
225: // Now, the character we just got is in the correct TAPE parity. If we
226: // are reading tape in EVEN parity, we have to FLIP THE CHECK BIT!
227:
228: if((Channel->ChUnitType->Get().ToInt() & 2) == 0) { // B as 2 bit - ODD
229:     // NO 2 bit -- EVEN!
230:     b.ComplementCheck(); ← 'if (Channel->ModeMode == 8)
231: }                                b.To6Bit() == 8) { b = BITC;
232:
233: // If we set the wm flag earlier, turn it on (and flip the check bit)
234:
235: if(wm) {
236:     b.SetWM();
237:     b.ComplementCheck();
238: }
239:
240: // Finally, tell the channel we have something to eat.
241:
242: Channel->ChannelStrobe(b);
243: }
244:
245: // Method for unit control. Mostly, just passes it off to the tape unit.
246: // Also sets ExtEndofTransfer, indicating we are done. (Will probably have
247: // to work in delays someday. Oh well)
248:
249: // Note that, interestingly, the TAU returns busy for many successful
250: // unit control operations.
251:
252: void TTapeTAU::DoUnitControl(BCD opmod) {
253:
254:     int d;
255:     int rc;
256:
257:     d = opmod.To6Bit();
258:
259:     if(TapeUnit == NULL) {
260:         tapestatus |= IOCHNOTREADY;
261:         Channel->ExtEndofTransfer = true;
262:         return;
263:     }
264:
```

```
265:     switch(d) {
266:
267:     case UNIT_BACKSPACE:
268:         if(!TapeUnit -> Backspace()) {
269:             tapestatus |= IOCHNOTREADY;
270:         }
271:         // tape indicate does NOT cause CONDITION in backspace operation!
272:         break;
273:
274:     case UNIT_SKIP:
275:         if(!TapeUnit -> Skip()) {
276:             tapestatus |= IOCHNOTREADY;
277:         }
278:         break;
279:
280:     case UNIT_REWIND:
281:         if(!TapeUnit -> Rewind()) {
282:             tapestatus |= IOCHNOTREADY;
283:         }
284:         break;
285:
286:     case UNIT_REWIND_UNLOAD:
287:         if(!TapeUnit -> Unload()) {
288:             tapestatus |= IOCHNOTREADY;
289:         }
290:         break;
291:
292:     case UNIT_WTM:
293:         if(!TapeUnit -> WriteTM()) {
294:             tapestatus |= IOCHNOTREADY;
295:         }
296:         // Write TM in odd parity sets Data Check
297:         if(Channel -> ChUnitType -> Get().ToInt() & 2) {
298:             tapestatus |= IOCHDATACHECK;
299:         }
300:         break;
301:
302:     case UNIT_SPACE:
303:         rc = TapeUnit -> Space();
304:         switch(rc) {
305:             case TAPEUNITEOF:
306:                 tapestatus |= IOCHCONDITION;
307:                 break;
308:             case TAPEUNITNOTREADY:
309:                 tapestatus |= IOCHNOTREADY;
310:                 break;
311:             case TAPEUNITERROR:
312:                 tapestatus |= (IOCHNOTREADY | IOCHCONDITION);
313:                 break;
314:             default:
315:                 break;
316:             }
317:             break;
318:
319:         default:
320:             tapestatus |= IOCHNOTREADY;
321:             break;
322:         }
323:
324:         FI729 -> Display();
325:         Channel -> ExtEndofTransfer = true;
326:         return;
327:     }
328:
329: // Private utility methods...
330:
```

```
331: // Utility method to write a character to the drive and collect status.
332:
333: bool TTapeTAU::DoOutputWrite(char c) {
334:
335:     int wanted_parity;
336:
337:     if(TapeUnit == NULL) {
338:         tapestatus |= IOCHNOTREADY;
339:         Channel -> ExtEndofTransfer = true;
340:         return(false);
341:     }
342:
343:     c &= 0x3f;                                // Bye bye WM & Check Bit.
344:
345:     // Check the unit type. B, which has a "2" bit, means odd.
346:
347:     wanted_parity = ((Channel -> ChUnitType -> Get().ToInt() & 2) != 0); // 1 Odd
348:
349:     if(tape_parity_table[c] != wanted_parity) {
350:         c ^= BITC;
351:     }
352:
353:     // Finally, tell the drive to write the character.           — if (c == 0) { assert(wanted_parity == 0); C = (BITA | BITC); }
354:
355:     return(TapeUnit -> Write(c));
356: }
357:
358: // Utility method to read a character from a tape drive. Sets appropriate
359: // status if the drive returns something other than an ordinary character.
360:
361: int TTapeTAU::DoInputRead() {
362:
363:     int c;
364:     bool wanted_parity;
365:
366:     c = TapeUnit -> Read();
367:
368:     if(c < 0) {
369:         Channel -> ExtEndofTransfer = true;
370:         if(c == TAPEUNITNOTREADY || c == TAPEUNITERROR) {
371:             tapestatus |= IOCHNOTREADY;
372:             return(c);
373:         }
374:         else if(c == TAPEUNITEOF) {
375:             tapestatus |= IOCHCONDITION;
376:             c = BCD_TM;                      // Continue on with parity check
377:         }
378:         else {
379:             return(c);
380:         }
381:     }
382:
383:     // Check what parity we want. We want odd parity if the device in the
384:     // instruction was "B" (has a 2 bit).
385:
386:     wanted_parity = ((Channel -> ChUnitType -> Get().ToInt() & 2) != 0);
387:
388:     assert(c >= 0 && c < 0x80);
389:     if(tape_parity_table[c] != wanted_parity) {
390:         tapestatus |= IOCHDATACHECK;
391:     }
392:
393:     return(c);
394: }
395:
396: // Return status at end of operation
```

```
397:  
398: int TTapeTAU::StatusSample() {  
399:     if(TapeUnit != NULL) {  
400:         FI729 -> Display();  
401:     }  
402:  
403:     if(Channel -> ChWrite -> State()) {  
404:         if(TapeUnit == NULL) {  
405:             tapestatus |= IOCHNOTREADY;  
406:         }  
407:         else {  
408:             DEBUG("TAU Initiating WriteIRG", 0);  
409:             TapeUnit -> WriteIRG();  
410:         }  
411:     }  
412:  
413:     if(TapeUnit != NULL && (tapestatus & IOCHCONDITION)) {  
414:         TapeUnit -> ResetIndicate();  
415:         Channel -> SetTapeIndicate(); } tapestatus |= IOCHCONDITION;  
416:     }  
417:     return(Channel -> GetStatus() | tapestatus);  
418: }
```

// TapeUnit -> TapeIndicate()



```
67:     bool OutputRequest;           // Not in real machine - for output
68:                                         // (co-routines)
69:     bool LastInputCycle;          // Indicate last cycle of input
70:     bool EndofRecord;            // Indicate record has ended
71:     bool TapeIndicate;           // If a tape operation set TI
72:     T1410IODevice *CurrentDevice; // Ptr to device doing transfer
73:
74:     enum TapeDensity {
75:         DENSITY_200_556 = 0, DENSITY_200_800 = 1, DENSITY_556_800 = 2
76:     } TapeDensity;
77:
78:
79: // Methods
80:
81: T1410Channel(                                // Constructor
82:     TAddressRegister *Addr,
83:     TLabel *LampInterlock,
84:     TLabel *LampRBCInterlock,
85:     TLabel *LampRead,
86:     TLabel *LampWrite,
87:     TLabel *LampOverlap,
88:     TLabel *LampNotOverlap,
89:     TLabel *LampNotRead,
90:     TLabel *LampBusy,
91:     TLabel *LampDataCheck,
92:     TLabel *LampCondition,
93:     TLabel *LampWLRecord,
94:     TLabel *LampNoTransfer
95: );
96:
97: // The following routine is called from T1410IODevice constructors
98: // to add devices into the Channel's device table.
99:
100: void AddIODevice(T1410IODevice *iodevice, int devicenumber);
101:
102: // Methods that do most of the Channel's real work.
103:
104: inline int SetStatus(int i) { return ChStatus = i; }
105: inline int GetStatus() { return ChStatus; }
106: bool IsTapeIndicate() { return TapeIndicate; }
107: void SetTapeIndicate() { TapeIndicate = true; }
108: void ResetTapeIndicate() { TapeIndicate = false; }
109:
110: inline int GetDeviceNumber() {
111:     return ChUnitType -> Get().ToInt() & 0x3f;
112: }
113: T1410IODevice *SetCurrentDevice() {
114:     return CurrentDevice = Devices[GetDeviceNumber()];
115: }
116: T1410IODevice *GetCurrentDevice() { return CurrentDevice; }
117: inline int GetUnitNumber() {
118:     return ChUnitNumber -> Get().ToAscii() - '0';
119: }
120:
121: void DoOutput(TAddressRegister *addr);        // Channel to Device
122: void DoInput(TAddressRegister *addr);          // Device to Channel - transfer
123: bool ChannelStrobe(BCD ch);                  // Device to Channel - Request
124: void DoUnitControl(BCD opmod);                // Unit control
125: void DoOverlap();                            // Overlap cycle processing
126:
127: // Channel register methods
128:
129: inline bool GetR1Status() { return R1Status; }
130: inline bool GetR2Status() { return R2Status; }
131: inline void ResetR1() { R1Status = false; }
```

```
133:     inline void ResetR2() { R2Status = false; };
134:     BCD SetR1(BCD b);
135:     BCD SetR2(BCD b);
136:     BCD MoveR1R2();
137: }
138:
139: // Class declaration for IO Devices. This is an *abstract* class,
140: // which is used to derive a class for each kind of IO device (console,
141: // reader, punch, tape, disk, etc. The Unit Control function is the
142: // only one that is not pure virtual. We provide a default for it because
143: // most devices don't implement unit control
144:
145: class T1410IODevice : public TObject {
146:
147: protected:
148:     T1410Channel *Channel; // Channel device is attached to
149:
150: public:
151:     T1410IODevice(int devicenum, T1410Channel *Channel); // Constructor
152:     virtual int Select() = 0; // Start an operation
153:     virtual int StatusSample() = 0; // Return device status
154:     virtual void DoOutput() = 0; // Channel -> Device
155:     virtual void DoInput() = 0; // Device -> Channel
156:     virtual void DoUnitControl(BCD opmod); // Unit Control
157: }
158:
159: #endif
```

```
1: //-----
2: #include <vc1.h>
3: #pragma hdrstop
4:
5: #include "UBCD.H"
6: #include "UI1410CPUT.H"
7: #include "UI1410CHANNEL.h"
8:
9: //-----
10: #pragma package(smart_init)
11:
12: #include <assert.h>
13: #include <stdio.h>
14:
15: #include "UI1410DEBUG.H"
16: #include "UI1410INST.H"
17: #include "UI1415IO.H"
18: #include "UI1415CE.H"
19:
20: // Implementation of I/O Channel Class
21:
22: // Constructor.  Initializes state
23:
24: T1410Channel::T1410Channel(
25:     TAddressRegister *Addr,
26:     TLabel *LampInterlock,
27:     TLabel *LampRBCInterlock,
28:     TLabel *LampRead,
29:     TLabel *LampWrite,
30:     TLabel *LampOverlap,
31:     TLabel *LampNotOverlap,
32:     TLabel *LampNotReady,
33:     TLabel *LampBusy,
34:     TLabel *LampDataCheck,
35:     TLabel *LampCondition,
36:     TLabel *LampWLRecord,
37:     TLabel *LampNoTransfer ) {
38:
39:     int i;
40:
41:     ChStatus = 0;
42:     TapeDensity = DENSITY_200_556;
43:     R1Status = R2Status = false;
44:
45:     ChStatusDisplay[IOLAMPNOTREADY] = LampNotReady;
46:     ChStatusDisplay[IOLAMPBUSY] = LampBusy;
47:     ChStatusDisplay[IOLAMPDATACHECK] = LampDataCheck;
48:     ChStatusDisplay[IOLAMPCONDITION] = LampCondition;
49:     ChStatusDisplay[IOLAMPNOTTRANSFER] = LampNoTransfer;
50:     ChStatusDisplay[IOLAMPWLRECORD] = LampWLRecord;
51:
52:     // Generally, the channel latches are *not* reset by Program Reset
53:
54:     ChInterlock = new TDisplayLatch(LampInterlock, false);
55:     ChrBCInterlock = new TDisplayLatch(LampRBCInterlock, false);
56:     ChRead = new TDisplayLatch(LampRead, false);
57:     ChWrite = new TDisplayLatch(LampWrite, false);
58:     ChOverlap = new TDisplayLatch(LampOverlap, false);
59:     ChNotOverlap = new TDisplayLatch(LampNotOverlap, false);
60:
61:     // Generally, the channel registers are *not* reset by Program Reset
62:
63:     ChOp = new TRegister(false);
64:     ChUnitType = new TRegister(false);
65:     ChUnitNumber = new TRegister(false);
66:     ChR1 = new TRegister(false);
```

```
67:     ChR2 = new TRegister(false);
68: 
69:     // Set up a pointer to the associated address register
70: 
71:     ChAddr = Addr;
72: 
73:     // Clear the device table
74: 
75:     for(i=0; i < 64; ++i) {
76:         Devices[i] = NULL;
77:     }
78: 
79:     CurrentDevice = NULL;
80:     MoveMode = LoadMode = false;
81:     ExtEndofTransfer = IntEndofTransfer = false;
82:     CycleRequired = false;
83:     InputRequest = false;
84:     OutputRequest = false;
85:     LastInputCycle = false;
86:     EndofRecord = false;
87:     TapeIndicate = false;
88: }
89: 
90: // Channel Register methods
91: 
92: BCD T1410Channel::SetR1(BCD b) {
93:     ChR1 -> Set(b);
94:     R1Status = true;
95:     return(b);
96: }
97: 
98: BCD T1410Channel::SetR2(BCD b) {
99:     ChR2 -> Set(b);
100:    R2Status = true;
101:    return(b);
102: }
103: 
104: // Move data from register 1 to register 2
105: // Clearing register 1 in the process.
106: 
107: BCD T1410Channel::MoveR1R2() {
108:     ChR2 -> Set(ChR1 -> Get());
109:     R1Status = false;
110:     R2Status = true;
111:     return(ChR2 -> Get());
112: }
113: 
114: // Method to add a new device to the channel's device table.
115: // Typically this is called from the T1410IODevice base class constructor
116: 
117: void T1410Channel::AddIODevice(T1410IODevice *iodevice, int devicenumber) {
118:     assert(Devices[devicenumber] == NULL);
119:     Devices[devicenumber] = iodevice;
120: }
121: 
122: // Channel is reset during ComputerReset
123: 
124: void T1410Channel::OnComputerReset()
125: {
126:     Reset();
127: }
128: 
129: void T1410Channel::Reset() {
130:     ChStatus = 0;
131:     R1Status = R2Status = false;
132:     MoveMode = LoadMode = false;
```

```
133:     ExtEndofTransfer = IntEndofTransfer = false;
134:     CycleRequired = false;
135:     InputRequest = false;
136:     LastInputCycle = false;
137:     EndofRecord = false;
138:     ChInterlock -> Reset();
139:     ChRBCTInterlock -> Reset();
140:     ChRead -> Reset();
141:     ChWrite -> Reset();
142:     ChOverlap -> Reset();
143:     ChNotOverlap -> Reset();
144: }
145:
146: // Channel is not reset during Program Reset
147:
148: void T1410Channel::OnProgramReset()
149: {
150:     // Channel not affected by Program Reset
151: }
152:
153: // Display Routine.
154:
155: void T1410Channel::Display() {
156:
157:     int i;
158:
159:     ChStatusDisplay[IOLAMPNOTREADY] -> Enabled =
160:         ((ChStatus & IOCHNOTREADY) != 0);
161:     ChStatusDisplay[IOLAMPBUSY] -> Enabled =
162:         ((ChStatus & IOCHBUSY) != 0);
163:     ChStatusDisplay[IOLAMPDATACHECK] -> Enabled =
164:         ((ChStatus & IOCHDATACHECK) != 0);
165:     ChStatusDisplay[IOLAMPCONDITION] -> Enabled =
166:         ((ChStatus & IOCHCONDITION) != 0);
167:     ChStatusDisplay[IOLAMPWLRECORD] -> Enabled =
168:         ((ChStatus & IOCHWLRECORD) != 0);
169:     ChStatusDisplay[IOLAMPNOTTRANSFER] -> Enabled =
170:         ((ChStatus & IOCHNOTTRANSFER) != 0);
171:
172:     for(i=0; i <= 5; ++i) {
173:         ChStatusDisplay[i] -> Repaint();
174:     }
175:
176:     // Although in most instances the following would be redundant,
177:     // because these objects are also on the CPU display list, we include
178:     // them here in case we want to display a channel separately.
179:
180:     ChInterlock -> Display();
181:     ChrBCTInterlock -> Display();
182:     ChRead -> Display();
183:     ChWrite -> Display();
184:     ChOverlap -> Display();
185:     ChNotOverlap -> Display();
186: }
187:
188: // Channel Lamp Test
189:
190: void T1410Channel::LampTest(bool b)
191: {
192:     int i;
193:
194:     // Note, we don't have to do anything to the TDisplayLatch objects in
195:     // the channel for lamp test. They will take care of themselves on a
196:     // lamp test.
197:
198:     if(!b) {
```

```

199:         for(i=0; i <= 5; ++i) {
200:             ChStatusDisplay[i] -> Enabled = true;
201:             ChStatusDisplay[i] -> Repaint();
202:         }
203:     }
204:     else {
205:         Display();
206:     }
207: }
208:
209: // Channel Output to Device
210: // Conditions on Entry: B data register (B_REG) contains character to output
211: // Channel Register 1 (E1 or F1) should be empty.
212:
213: void T1410Channel::DoOutput(TAddressRegister *addr) {
214:
215:     BCD tempbcd;
216:
217:     CPU -> CycleRing -> Set(this == CPU -> Channel[CHANNEL1] ? CYCLE_E : CYCLE_F);
218:
219:     // Read out the next character. Also, check for a storage wrap, which will
220:     // end the transfer after this character.
221:
222:     *(CPU -> STAR) = *addr;
223:     CPU -> Readout();
224:     CPU -> StorageWrapCheck(+1);
225:     tempbcd = CPU -> B_Reg -> Get();
226:
227:     // If we have a GMWM, then we are done transferring data from memory
228:     // Otherwise, Clear the WM from the input data if in Move mode, and
229:     // then (regardless of mode) transfer the data to Channel register R1
230:     // Also, if we actually have read a character out, check for storage
231:     // wrap, which also ends the transfer.
232:
233:     if(tempbcd.TestGMWM()) {           68
234:         IntEndofTransfer = true;
235:         EndofRecord = true;
236:     }
237:     else {
238:         if(MoveMode) {
239:             tempbcd.ClearWM();
240:             tempbcd.SetOddParity();
241:         }
242:         SetR1(tempbcd);
243:         if(CPU -> StorageWrapLatch){    ←
244:             IntEndofTransfer = true;
245:             EndofRecord = true;
246:         }
247:     }
248:
249:     // Now, if we have anything for the device, send it. We continue
250:     // doing this for up to two characters, in case R1 and R2 are both
251:     // full. We have to do it this way so that if, at the end, we set
252:     // ExtEndofTransfer, all of the data will have been sent.
253:
254:     while(GetR1Status() || GetR2Status()) {
255:         if(GetR1Status() & !GetR2Status()) {
256:             MoveR1R2();
257:         }
258:         CurrentDevice -> DoOutput();
259:         ResetR2();
260:     }
261:
262:     // Advance the appropriate address register to the next location.
263:     // (Storage wrap was detected earlier, as necessary!).
264:

```

(ChOverlap → State 11)  
CPU → Op-Mod-Reg → Get().To6Bit  
!OP-MOD-SYMBOL-X )

```

265:     addr->Set(CPU->STARMod(+1)); } else addr->Set(CPU->STARMod(1));
266:
267: // Now that the data is all sent, if we are at IntEndofTransfer (either
268: // From a GMWM or from storage wrap), also set ExtEndofTransfer so that
269: // we quit. Note that the device may have already set ExtEndofTransfer!
270:
271: if(IntEndofTransfer) {
272:     ExtEndofTransfer = true;
273: }
274:
275: }
276:
277: // Channel Input Processing (shared overlap/not overlap code)
278:
279: void T1410Channel::DoInput(TAddressRegister *addr) {
280:
281:     CycleRequired = false; // Reset cycle required
282:     InputRequest = false; // Reset co-routine flag
283:
284:     if(!IntEndofTransfer) {
285:         *(CPU->STAR) = *addr; // Copy memory address
286:         CPU->Readout(); // Get existing memory
287:         CPU->AChannel->Select(); // Gate A channel approp.
288:         this == CPU->/Channel[CHANNEL1] ?
289:             TAChannel::A_Channel_E : TAChannel::A_Channel_F );
290:     }
291:
292:     // If no data from device, the end is near...
293:
294:     if(!(GetR1Status() || GetR2Status())) {
295:         LastInputCycle = true;
296:     }
297:
298:     // If B GMWM, we are typically done storing data (unless this
299:     // is read to end of core, which ignores GMWM). Unless, of course,
300:     // we have already hit end of record.
301:
302:     // Note that we can only check the op mod if we are NOT overlapped
303:     // (Note that the "to end of core" mods are all not overlapped)
304:
305:     if(!EndofRecord && CPU->B_Reg->Get().TestGMWM()) {
306:         if(ChOverlap->State() ||
307:             (CPU->Op_Mod_Reg->Get().To6Bit() != OP_MOD_SYMBOL_DOLLAR) {
308:                 addr->Set(CPU->STARMod(1)); // Bump address register
309:                 EndofRecord = true;
310:             }
311:     }
312:
313:     // If no more input, or we hit GMWM,
314:     // set internal end of transfer, but keep on accepting characters
315:     // until External end of transfer.
316:
317:     if(LastInputCycle || EndofRecord) {
318:         IntEndofTransfer = true; // If(!IntEndofTransfer) {
319:         if(ExtEndofTransfer) { // addr->Set(CPU->STARMod(1));
320:             return; } }
321:     }
322:
323:
324:     // If we get here, we are either ending, or we are still storing data
325:
326:     if(!LastInputCycle && !IntEndofTransfer) { // Still storing input?
327:
328:         // If parity is no good, set data check...
329:
330:         if(!CPU->AChannel->Select().CheckParity()) {

```

```
331:             SetStatus(GetStatus() | IOCHDATACHECK);
332:
333:             // If asterisk insert, store an asterisk!
334:
335:             if(FI1415CE -> AsteriskInsert -> Checked) {
336:                 Chr2 -> Set(BCD_ASTERISK);
337:                 CPU -> AChannel -> Select(
338:                     this == CPU -> Channel[CHANNEL1] ?
339:                         TAChannel::A_Channel_E : TAChannel::A_Channel_F);
340:                 }
341:             } // End initial parity check
342:
343:             // Store the data (perhaps with a parity error!
344:
345:             CPU -> Store(CPU -> AssemblyChannel -> Select(
346:                 (MoveMode ?
347:                     TAAssemblyChannel::AsmChannelWMB :
348:                     TAAssemblyChannel::AsmChannelWMA),
349:                     TAAssemblyChannel::AsmChannelZonesA,
350:                     false,
351:                     TAAssemblyChannel::AsmChannelSignNone,
352:                     TAAssemblyChannel::AsmChannelNumA) );
353:
354:             // If we have bad data, but not asterisk insert, STOP
355:             // One good way: Run FORTRAN w/o Asterisk Insert! ;)
356:
357:             if(!CPU -> AChannel -> Select().CheckParity()) {
358:                 CPU -> AChannelCheck -> SetStop("Data Check, No Asterisk Insert!");
359:                 return;
360:             }
361:
362:             ResetR2();                                // Reset Channel Data
363:             if( CPU -> StorageWrapCheck(+1)) {          // If storage wrap -- done
364:                 IntEndofTransfer = true;
365:                 EndofRecord = true;
366:             }
367:             else {
368:                 addr -> Set(CPU -> STARMod(1));        // Bump address register
369:             }
370:
371:         } // End, !LastInputCycle
372:
373:         // In the real world, the device keeps reading data, and will
374:         // then set CycleRequired. (In fact, the console matrix will
375:         // call ChannelStrobe to actually do this when a key is pressed).
376:         // But most of our input devices in the emulator are co-routines,
377:         // so we have to call them back to give them a chance to strobe
378:         // the channel again. Also, in the real world, a device would
379:         // keep reading input data after the channel set Internal End of
380:         // Transfer -- we have to give the input co-routine a way to
381:         // finish reading it's record, even if we are no longer storing
382:         // it because we hit a GMWM or wrapped storage.
383:
384:         CurrentDevice -> DoInput();
385:
386:     }
387:
388:     bool T1410Channel::ChannelStrobe(BCD ch) {
389:
390:         // If R1 has data already, move it to R2.
391:         // (In the emulator, that should probably never happen!)
392:
393:         if(GetR1Status()) {
394:             MoveR1R2();
395:         }
396:
```

```
397: // Load R1, and copy to R2 if there is room in R2.  
398:  
399: SetR1(ch);  
400: if(!GetR2Status()) {  
401:     MoveR1R2();  
402: }  
403:  
404: // If the channel has not already terminated the transfer,  
405: // ask to send the data to memory.  
406:  
407: if(!IntEndofTransfer) {  
408:     CycleRequired = true;  
409: }  
410:  
411: InputRequest = true; // Force co-routine call  
412: return true;  
413: }  
414:  
415: // Channel Unit Control just passes it off to the device.  
416:  
417: void T1410Channel::DoUnitControl(BCD opmod) {  
418:     GetCurrentDevice() -> DoUnitControl(opmod);  
419:     if(ExtEndofTransfer) {  
420:         IntEndofTransfer = true;  
421:     }  
422:     return;  
423: }  
424: // Channel Overlap Processing  
425: //  
426:  
427: void T1410Channel::DoOverlap() {  
428:  
429:     if((ChOp -> Get().ToInt() & OP_MOD_SYMBOL_W) == OP_MOD_SYMBOL_W) {  
430:         if(ExtEndofTransfer) {  
431:             ~ChOverlap -> Reset();  
432:             ~SetStatus(GetCurrentDevice() -> StatusSample());  
433:             return;  
434:         }  
435:         if(OutputRequest) {  
436:             CycleRequired = OutputRequest = false;  
437:             ~DoOutput(ChAddr);  
438:         }  
439:         return;  
440:     }  
441:     else if((ChOp -> Get().ToInt() & OP_MOD_SYMBOL_R) == OP_MOD_SYMBOL_R) {  
442:         if(ExtEndofTransfer) {  
443:             ChOverlap -> Reset();  
444:             SetStatus(GetCurrentDevice() -> StatusSample());  
445:             if(CycleRequired || GetR2Status() || !EndofRecord) {  
446:                 SetStatus(GetStatus() | IOCHWLRECORD);  
447:             }  
448:             return;  
449:         }  
450:         if(CycleRequired || InputRequest) {  
451:             CycleRequired = InputRequest = false;  
452:             DoInput(ChAddr);  
453:         }  
454:         return;  
455:     }  
456: }  
457:  
458: □  
459:  
460: // Class T1410IODevice implementation. This is an *abstract* base class,  
461: // intended to be used to derive actual I/O devices  
462:
```

```
463: T1410IODevice::T1410IODevice(int devicenumber, T1410Channel *Ch) {
464:     Channel = Ch;
465:     Ch -> AddIODevice(this,devicenumber);
466: }
467:
468: // Method to handle Unit Control for those devices which don't have unit
469: // control (most of them)
470:
471: void T1410IODevice::DoUnitControl(BCD opmod) {
472:     Channel -> SetStatus(Channel -> GetStatus() | IOCHNOTREADY);
473:     DEBUG("Unit control not implemented for device",0);
474:     return;
475: }
476:
477:
478:
479: // And, finally, the 1410 IO Instruction Routines. We implement them
480: // here to keep the I/O stuff together!
481:
482: // Move and Load mode (M and L) Instructions.
483:
484: // Note: The channel selected by the CPU is known to be available, as the
485: // interlock test was passed during instruction readout at I3.
486: // IOChannelSelect indicates the selected channel.
487: // Channel -> ChUnitType has Device Type (e.g. 'T' for console)
488: // Channel -> ChUnitNumber has Unit Number
489:
490: void T1410CPU::InstructionIO() {
491:
492:     BCD opmod;
493:     T1410Channel *Ch = Channel[IOChannelSelect];
494:
495:     opmod = (Op_Mod_Reg -> Get()).To6Bit();
496:     assert(!(Ch -> ChInterlock -> State()));
497:
498:     // Reset the channel, then set Channel Interlock
499:
500:     Ch -> Reset();
501:     Ch -> ChInterlock -> Set();
502:
503:     // Set Move mode or Load mode, appropriately
504:
505:     if((Op_Reg -> Get()).To6Bit() == OP_IO_MOVE) {
506:         Ch -> MoveMode = true;
507:     }
508:     else {
509:         Ch -> LoadMode = true;
510:     }
511:
512:     // Start things out, depending on op modifier.
513:
514:     switch(opmod.ToInt()) {
515:
516:     case OP_MOD_SYMBOL_R:
517:     case OP_MOD_SYMBOL_DOLLAR:
518:         Ch -> ChRead -> Set();
519:         break;
520:
521:     case OP_MOD_SYMBOL_W:
522:     case OP_MOD_SYMBOL_X:
523:         Ch -> ChWrite -> Set();
524:         break;
525:
526:     default:
527:         InstructionCheck ->
528:             SetStop("Instruction Check: Invalid I/O d-character");
```

```
529:         return;
530:     } // End switch on op modifier
531:
532:     // See if there is a device for this device number. If not,
533:     // return not ready.
534:
535:     if(Ch -> GetCurrentDevice() == NULL) {
536:         Ch -> SetStatus(IOCHNOTREADY);
537:         IRingControl = true;
538:         return;
539:     }
540:
541:     // Start up the I/O, do initial status check (Status Sample A)
542:     // Just return if the status is not 0.
543:
544:     Ch -> SetStatus(Ch -> GetCurrentDevice() -> Select());
545:     if(Ch -> GetStatus() != 0) {
546:         IRingControl = true;
547:         return;
548:     }
549:
550:     // If OK so far, set Overlap/NotOverlap
551:
552:     // If overlapped, copy the BAR to the associated channel register,
553:     // and start the I/O operation before returning.
554:
555:     if(CPU -> IOOverlapSelect) {
556:         ~Ch -> ChOverlap -> Set();
557:         *(Ch -> ChAddr) = *B_AR;
558:         ~Ch -> ChOp -> Set(opmod);
559:         if((opmod.ToInt() & OP_MOD_SYMBOL_W) == OP_MOD_SYMBOL_W) {
560:             ~Ch -> DoOutput(Ch -> ChAddr);
561:         }
562:         else if((opmod.ToInt() & OP_MOD_SYMBOL_R) == OP_MOD_SYMBOL_R) {
563:             ~Ch -> GetCurrentDevice() -> DoInput();
564:         }
565:         IRingControl = true;
566:         return;
567:     }
568:
569:     // If we get here, we are not overlapped.
570:
571:     ~Ch -> ChNotOverlap -> Set();
572:
573:     if((opmod.ToInt() & OP_MOD_SYMBOL_W) == OP_MOD_SYMBOL_W) {
574:         CPU -> Display();
575:         while(!Ch -> ExtEndofTransfer) {
576:             ~Ch -> DoOutput(B_AR);
577:         }
578:         ~Ch -> ChNotOverlap -> Reset();
579:         ~Ch -> SetStatus(Ch -> GetCurrentDevice() -> StatusSample());
580:         *IRingControl = true;
581:         return;
582:     }
583:     else if((opmod.ToInt() & OP_MOD_SYMBOL_R) == OP_MOD_SYMBOL_R) {
584:
585:         // Input processing continues so long as not External End from device
586:
587:         CPU -> Display();
588:         ~Ch -> GetCurrentDevice() -> DoInput();
589:         while(!Ch -> ExtEndofTransfer) {
590:
591:             // If no input, just wait here (not overlapped -- stuck here)
592:
593:             while(!(Ch -> CycleRequired || Ch -> InputRequest ||
594:                   Ch -> ExtEndofTransfer)) {
```

```
595:             Application -> ProcessMessages();
596:             // sleep(10);
597:             continue;
598:         }
599:
600:         _Ch -> DoInput(B_AR);
601:
602:     } // End, not External End of Transfer
603:
604:     assert(Ch -> ExtEndofTransfer);
605:
606:     // Here comes a bit of a kludge. Except for the console, whose
607:     // inquiry request key in the emulator actually works in real time,
608:     // the device will set External End of Transfer before the channel
609:     // has a chance to actually read the next character. This causes
610:     // a WLR indication. So, here, if the device set External End of
611:     // Transfer, and there is no data from it, we call the Channel for
612:     // input one more time. Because the device has no data, the
613:     // Channel DoInput method will set LastInputCycle, therefore
614:     // setting IntEndofTransfer, and return here without calling the
615:     // device again.
616:
617:     ~if(Ch -> ChUnitType -> Get().To6Bit() != CONSOLE_IO_DEVICE &&
618:         ~(Ch -> GetR1Status() || Ch -> GetR2Status())) {
619:         ~Ch -> DoInput(B_AR);
620:     }
621:
622:     // If, at the end, things do not match up ==> Wrong Length Record
623:     // (Note that we do *not* test InputRequest here!)
624:
625:     ~Ch -> SetStatus(Ch -> GetCurrentDevice() -> StatusSample());
626:     ~if(Ch -> CycleRequired || Ch -> GetR2Status() || !Ch -> EndofRecord) {
627:         ~Ch -> SetStatus(Ch -> GetStatus() | IOCHWLRECORD);
628:     }
629:
630:     // And all done - continue with Instructions
631:
632:     ~Ch -> ChNotOverlap -> Reset();
633:     XIRingControl = true;
634:     _return;
635: }
636: }
637:
638:
639: // Unit control (U) Instruction. Basically, it just passes it on to
640: // the appropriate unit. The rest of the code is similar to the normal
641: // I/O instructions.
642:
643: void T1410CPU::InstructionUnitControl() {
644:
645:     BCD opmod;
646:     T1410Channel *Ch = Channel[IOChannelSelect];
647:
648:     // We assume that the channel is valid, and not interlocked.
649:
650:     assert(Ch != NULL);
651:     assert(!Ch -> ChInterlock -> State());
652:
653:     // The unit control function is defined by the d-character.
654:
655:     opmod = Op_Mod_Reg -> Get().To6Bit();
656:
657:     // Start of an I/O operation. Reset the Channel, and set Interlocked.
658:     // Check that the device requested exists, and select it. If anything
659:     // goes wrong, return.
660:
```

```
661:     Ch -> Reset();
662:     Ch -> ChInterlock -> Set();
663:     if(Ch -> GetCurrentDevice() == NULL) {
664:         Ch -> SetStatus(IOCHNOTREADY);
665:         IRingControl = true;
666:         return;
667:     }
668:     Ch -> SetStatus(Ch -> GetCurrentDevice() -> Select());
669:     if(Ch -> GetStatus() != 0) {
670:         IRingControl = true;
671:         return;
672:     }
673:
674: // Set Overlap/Not Overlap in progress, appropriately. If Overlapped,
675: // fire things up, then return to the instruction stream.
676:
677: if(CPU -> IOOverlapSelect) {
678:     Ch -> ChOverlap -> Set();
679:     // To Do: Start the I/O operation... ←
680:     IRingControl = true;
681:     return;
682: }
683: else {
684:     Ch -> ChNotOverlap -> Set();
685: }
686:
687: // If we get here, we are not overlapped...
688:
689: Ch -> DoUnitControl(opmod);
690:
691: // All we can do now is wait...
692:
693: while(!Ch -> ExtEndofTransfer) {
694:     Application -> ProcessMessages();
695:     // sleep(10);
696: }
697:
698: // Finish up the operation...
699:
700: Ch -> ChNotOverlap -> Reset();
701: Ch -> SetStatus(Ch -> GetCurrentDevice() -> StatusSample());
702: IRingControl = true;
703: return;
704: }
705:
```

```
1: //-----
2: #ifndef UIPRINTERH
3: #define UIPRINTERH
4: //-----
5:
6: #define PRINTMAXFORM    1024
7:
8: //  Printer Adapter Unit (1414)
9:
10: class TPrinter : public TObject {
11:
12: protected:
13:
14:     int PrintStatus;                      // Printer Status for Channel
15:     bool Ready;                          // True if ready to go
16:     TBusyDevice *BusyEntry;              // Used to set delays
17:     int BufferPosition;                 // Current output column
18:     int SkipLines;                     // Deferred Skip in lines
19:     int SkipChannel;                  // Deferred Skip to Channel
20:
21:     int FormLength;                   // Length of current form
22:     int FormLine;                     // Current line in form
23:     enum Forms {
24:         FORM_SCREEN, FORM_FILE, FORM_PRINT
25:     };
26:
27:     TFileStream *ccfd;                // Carriage Control File
28:     int CarriageTape[PRINTMAXFORM];   // Carriage tape data
29:     char ccline[MAXLINE];
30:
31:     char FileName[MAXPATH];          // File name, if to file
32:     TFileStream *fd;                // File FDs for print, CC file
33:
34: public:
35:
36:     TPrinter(int devicenum, T1410Channel *Channel);
37:
38:     inline bool IsReady() { return(Ready); }
39:     inline bool IsBusy() { return BusyEntry -> TestBusy(); }
40:
41:     bool Start();                   // Called from UI Start Button
42:     void Stop();                   // Called from UI Stop Button
43:
44:     bool CarriageRestore();        // Carriage to Channel 1
45:     void CarriageStop();          // Called from UI Carriage Stop
46:     bool CarriageSpace();         // Space to next line
47:
48:     virtual int Select();          // Channel Select
49:     virtual void DoOutput();      // Character to output
50:     virtual void DoInput();       // NOP on this device
51:     virtual int StatusSample();   // End of I/O status sample
52:     virtual void DoUnitControl(BCD opmod); // CC operation
53:
54:     void DoOutputChar(BCD c);    // Send one char of output
55:     // TODO bool PrintAscii(char c); // Output to a file or printer
56:
57:     bool SetCarriageTape(char *tapefile); // Set up carriage control
58:     void SetCarriageDefault();     // Set default CC tape
59:     bool CarriageChannelTest(int cchannel); // True if this line has chan.
60:
61: private:
62:
63:     bool GetCarriageLine(TFileStream *fs, char *line);
64:     bool ParseCarriageLine(char *line, char **elements);
65: };
66:
```

```
67: #endif  
68:
```

T-PRINTER ::

# of bits CAPTURED CHANNEL 1

1	1
2	2
3	4
4	8
5	16
6	32
7	64
8	128
9	256
10	512
11	1024
12	2048

University Club

```

TPrinter : public TObject {
protected:
    char filename [MAXPATH];
    // path to file if sending to file
    IFileStream *fd, *ccfd;
    // file descriptor if sending to file
    char print-buffer [132];
}

enum forms {
    FORM-SCREEN, FORM-FILE, FORM-PRINT
};

int form-length;                                // length of form, in lines
int carriage-tape [424];
int form-line;                                  // line in form
bool ready;
IBusyDevice *BusyEntry;
int printstatus;
char cc-line [MAXLINE];
int deferred-space;                            >
int buffer-position;
int skip-lines;
int skip-channel;
}

```

## TPrinter::

public:

✓ inline bool IsReady() { return ready; }  
✓ inline bool IsBusy() { return BusyEntry > TextBusy(); }  
✓ void Start();  
✓ bool CarriageRestore();  
✓ void CarriageStop();  
✓ bool CarriageSpan();  
✓ void Stop();  
...

priv +

✓ GetCarriageLine(TFileStream  
\*ccfd, char \*line);  
✓ ParseCarryLine(char \*line,  
char \*\*element);

✓ void ControlCarriage(BCD dchar) <sup>✓ bool CarriageChannelTest(  
int channel);</sup>  
✓ bool CarriageSkip(int lines, int chan);

✓ virtual int Select();  
✓ virtual void DoOutput();  
✓ virtual void DoInput();  
✓ virtual int StatusSample();  
✓ virtual void DoUnitControl(BCD opmod);  
✓ void DoOutputChar(BCD c);

✓ bool SetCarriageTape(char \*topicfile);  
✓ void SetCarriageDefault();

X TPrinter::int devicenom, TI410Channel \*Channel);

TPrinter ::

bop) Print Ascii (char c);

TPrinter::TPrinter(int devicenum, T1410Channel \*channel) :

T1410IODevice(devicenum, channel) {

filename [0] = '\0';

fd = NULL;

forms = FORM-SCREEN;

SetCarriageDefault();

ready = true;

printstatus = 0;

BusyEntry = new TBusyDevice();

ccFd = NULL;

ccLine [0] = '\0';

~~deford-space = 0;~~

buffer-position = 0;

~~skip-times = skip-channel = 0;~~

```
bool TPrinter::Start() {  
    ready = true;  
}
```

TPrinter::

```
bool CarriageRestore() {  
    return (CarriageSkip(0, 1));  
}  
for (line = 0; line < form.length(); ++line) {  
    CarriageSpace();  
    if (CarriageChannelTest(CARRIAGE_CHANNEL - 1)) {  
  
  
}  
return (ready = false); // wrong carriage
```

~~3~~

```
bool TPrinter::CarriageChannelTest(int c) {  
    return ((carriage-type[form.line] & c) != 0);  
}
```

TPrinter::

```
void CarriageStop() {  
    ready = false;
```

~~3~~

```
TPrinter::  
void Stop() {  
    ready = false;  
}
```

~~3~~

```

int TPrinter :: Select() {
    if (!ready) {
        return (IOCHNOTREADY);
    }
    if (BusyEntry → IsBusy()) {
        return (IOCHBUSY);
    }
    ① → return (printstatus = φ); buffer-position = φ;
}

```

```

int TPrinter:: StatusSample() {
    return (Channel → GetStatus() | printstatus);
}

```

```

① { if (Channel → Read() → State())
      if (Channel → GetUnitNumber() != φ &
          != 1) {
          return (IOCHNOTREADY);
      }
}

```

```

void TPrinter DoInput() {
    printstatus = IOCHNOTREADY;
    return;
}

```

```

void TPrinter::DoOutput() {
    BCD ch-char;
    if (!ready) {
        printstatus |= IOCHNOTREADY;
        Channel → ExtEnd of Transfer = true;
        return;
    }

    ch-char = Channel → Chr2 → Get();
    if (buffer-position > 131) {
        printstatus |= IOCHWLRECORD;
        Channel → ExtEnd of Transfer = true;
        return;
    }

     →
    if (Channel → Move Mode) {
        if (Channel → GetUnitNumber() == 0) {
            DoOutputChar(ch-char);
        }
    } else {
        DoOutputChar(ch-char, TestWMC ?
                     BCD-1 : BCD-SPACE);
    }

    ++ buffer-position;
}

else {
    if (Channel → GetUnitNumber() == 0) {
        if (ch-char. Test WM) {
            DoOutputChar(BCD-W$);
            if (++ buffer-position > 131) {
                printstatus |= IOCHWLRECORD;
                Channel → ExtEnd of Transfer = true;
                return;
            }
        }
    }
}

```

else { (Load Mode)

if { (Unit 0)

    DoOutputChar(ch-char);  
    ++ buffer-position;

}

else { (Unit 1)

    DoOutputChar(ch-char, Test WMC)?  
        BCD\_WS : BCD-SPACE);  
    ++ buffer position;

}

} // Load mode

{ if (! ch-char.CheckParity ()) {

    PrintStatus 1 = IDCH DATACHECK;

}

if (buffer-position <= 131) {

    Channel → OutputRequest = true;

    Channel → Cycle Required = true;

}

else {

    Channel → ExtendEndOfTransfer = true;

    → returning

}

BusyDevice → SetBusy(1);

{ if (skip-lines != skip-channel) {

    0 1 2 3 4  
    ~ B C D E

        Carriage Skip (skip-lines, skip-channel)  
        skip-lines = skip-channel = 0;  
    } else {  
        Carriage Spool();

}

```
void TPrinter::DoOutputChar (BCD c) {
    word state;
    c = c.ToGBit(c);
    switch(forms) {
        case FORM_SCREEN:
            state = F114&3 → SendBCD(c);
            break;
        case FORM_FILE:
            state=PrintAscii(ToAscii(c));
            break;
    }
    if(!state) {
        printstate |= IOCH CONDITION;
    }
}
```

```
void TPrinter::DoUnitControl(BCD opmod) {
    Channel1 → UnitControl overlapped Busy = NULL;
    if (!ready) {
        printstatus1 = IOCHNOTREADY;
        Channel1 → ExtEnd of Transfer = true;
        return;
    }
    ControlCarriage (opmod);
    Channel1 → ExtEnd of Transfer = true;
    return;
}
```

```

Void TPrinter::ControlCarriage(BCD d-char) {
    int d-int;
    int num;

    d-int = d-char.TOBIT();
    num = d-int & BIT_NUM;

    if (num == 0) {
        return;
    }

    if (d-int & BIT_ZONE == 0) {
        if (num > 12) {
            return;
        }

        CarriageSkip(0, num);
        skip-lines = skip-channel = 0;
    }

    else if (d-int & BIT_ZONE == (BIT-A | BIT-B)) {
        skip-lines = 0;
        skip-channel = num;
    }

    else if (d-int & BIT_ZONE == BIT-B) {
        if (num > 3) {
            return;
        }

        CarriageSkip(num, 0);
    }

    else {
        if (num > 3) {
            return;
        }
    }
}

```

skip-lines = num;  
skip-channel = 0

```
bool TPrinter::CarriageSpace() {
    bool status;
    switch (forms) {
        case FORM_SCREEN:
            status = FALSE → NextLine();
            break;
        case FORM_FILE:
        case FORM_PRINT:
            status = Print ASCII ('\\n');
            break;
        default:
            assert (false);
            return status;
    }
    if (!status)
        return status;
    if (++form_line > form_length) {
        form_line = 1;
    }
    return status;
}
```

bool TPrinter::CarriageSkip(int chan, int spaces)

int chan-number;  
int line;

if (chan < 1 || chan > 12) {  
 return (ready = false);

}

chan-number = (1 << (chan - 1));

for (line = 0; line <= forms-length; ++ line) {

CarriageSpace();

if (CarriageChannelTest(chan-number)) {

return (true);

}

BusEntry → SetBusy();

}

return (ready = false);

}

if (spaces > 0 && spaces < 4) { // space request

while (spaces--) {

CarriageSpace();

}

return (true);

}



T Printer!!

boot SetCarriageDefault() {  
void

form-length = 66;

for (form-line = 1; form-line <= form-length; ++ form-line) {  
carriage-type [form-line] = 0;

}

form-line = 1;

carriage-type [4] = 1 1214181161

321641128 | 512 | 1024;

carriage-type [61] = 256;

63 = 2048;

form-line = 1;

}

int TPrinter::SetCarriageTape (char \*filename) ①

```
TFil Stream *ccfd;
char *elements [8];
int line[channel], i, n;
for (line = 0; line < 1024; ++ line) {
    carriage-tape [line] = 0;
}
if (filename == NULL || strlen (filename) == 0) {
    SetCarriageDefault ();
    return 0;
}
line = 0;
assert (ccfd == NULL);
try {
    ccf d = new TFile Stream (filename, fmOpenRead);
}
catch (EFOpen Error) {
    ccf d = NULL;
    return (-1);
}
do {
    if (!GetCarriageLine (ccfd, cc line)) {
        return (-1);
    }
    ++ line (cc line [0] != '#');
} until (cc line [0] != '#');
Parse CarriageLine (cc line, element );
```

SetCarryType ②

```

if (strcmp(element[0], "LENGTH")) {
    → return (-line); } ← delct ccfd;
form-length = atoi(element[1]);
if (form-length <= 8 || form-length > 1024) {
    → return (-line); } ← delct ccfd;
}
n = 0;
while (GetCarryLine(ccfd, ccline)) {
    ++line;
    if (ccline[0] == '*') {
        continue;
    }
    ParCarryLine(ccline, elements);
    if (strcmp(element[0]) != "CHAN" ||
        strcmp(element[2]) != "=")
        delct ccfd;
    return -line;
}
channel = atoi(element[1]);
if (channel < 1 || channel > 12) {
    delct ccfd;
    return -line;
}
for (i=3; element[i] != NULL; ++i) {
    if (*element[i] == '+') {
        n += atoi(element[i]+1);
    }
    else { n = atoi(element[i]); }
}

```

if ( $n < 1$  ||  $n >$  forms.length) { Set carriageTop(③)

carriageTop[n] =  
 $(1 \leq i \leq (n-1));$

}

} // for elements

}

formLine = 1;  
delete cfd;  
ccf1 = null;  
return (0);

}

```
bool TPrinter::GetCorrigLine (TFileStream *ccfd, char *line) {
    char *cp;
    if (ccfd == NULL) {
        return false;
    }
    for (cp = line; line - cp < MAXLINE; ++cp) {
        if (fd → Read(cp, 1) != 1) {
            return (false);
        }
        if (*cp == '\n') {
            *cp = '\0';
            return (true);
        }
    }
    line [MAXLINE] = '\0';
    return (false);
}
```

```
void TPrinter::ParseCorrigirLin (char *line, char **element) {
    int i = 0;
    while (*line) {
        element[i] = line;
        while (*++line && !iswhite(*line)) {      gobble non white
            }                                         do if null
        if (!*line) {                                set first white to null
            break;                                  gobble white space
            }
        *line++ = '\0';
        while (*++line && iswhite(*line)) {
            }
    }
}
```

## Input Flow

CPU → Instruction

Resets Channel

Set Interlock

Sets Local/Mac Mode

Sets Read/Write

Checks if device exists  
(NOTREADY if not)

① Device → Select()

| console

| Instruction

.

.

.

Device Select

Checks if unit exists  
(NOTREADY if not)

Tape (Resets transfer length to 0)

Initiate (if necessary)

Return initial state

Check initial state,  
quit if not 0

Set Overlap / Not Overlap

Overlap

Device → Do Input()

return

(Not Overlapped)

Device → Do Input

while (!done) (!getchanrequest)

Wait for Cycle Required  
or Input Request  
or Ext End of Transfer()

}

Channel → Do Input

Transfer to Storage

Device → Do Input()

}

Check Status

Check, get next ch  
Channel Stroke



Fill Channel Reg  
May Set Cycle Required  
Will Set Input Request  
(coroutine)

if

## In Overlap

for each channel ch

```
if (ch → ChOverlap → Stat()) {  
    if (ch → Ext End of Transfer) {  
        ch → SetStatus(ch → GetCurrentDevice() → StatusSample());  
        if (ch → Cycle Required || ch → GetR2Start()) ||  
            !ch → End of Record) {  
            ch → SetStatus(ch → GetStatus() | IOQHWRFLD());  
        }  
        ch → ChOverlap → Reset();  
    }  
    else if (ch → Cycle Required || ch → InputRequest) {  
        ch → DoInput (appropriate address register);  
        or Output  
    }  
}  
Output mode  
ch → DoOutput (appropriate address register)
```

25252

$X_1 = 40957$

$X_2 = 40772$

$FF = 40901$

$FF + 30$   $\checkmark$  ~~1000001001000~~ 40931

+40  $\checkmark$  ~~1000000~~ 40941

45  $\checkmark$  ~~100000~~ 40946

54  $\checkmark$  ~~1000000000~~ 40952

57  $\checkmark$  ~~10000000000~~ 40958

$\rightarrow$  58  $\checkmark$  ~~100000000000~~ 40959

64  $\checkmark$

65  $\checkmark$  ~~100000000000000~~

72  $\checkmark$

36551	36531	36511	36491	36471	36451	36431	36411
37003	37063	37043	37023	37003	36983	36963	36943

10777

36471  $\text{OK}$   
37003

36511  
37043 FAIUS

36491  
37023 FAIUS

36481 FAIUS  
37013

36476 FAIUS  
37008

36473 FAIUS  
37005 FAIUS

36472  $\text{OK}$   
37004

$36473 = 0x10 - \text{Read}$   
 $37005 = 0x40 - \text{Written} = \text{no bits} \rightarrow$

C+A

Load more, Even parity

$b1c32 = -31 \times 10^6$

More Mode

! Cyclic Required

! R2 Status

! Init Ext End of Transfer

(+) Ext End of Transfer.

(+) Top Indicator

! End of Record

R2 True

R2 False

Select off load

Write IRG or  $\frac{1}{2}4$

Read IRG or  $\frac{1}{2}2$

~~DoInput()~~

BSR must clear

DoInput

T.I.

Driver → DoInput()

1400 Ch → DoInput

3F → R1<sub>1,2</sub> → R2

9401

3F

9402 Read 3F → Channel Stat

9409 STORE

1410 TAPEDIN  
DoInput

DoInputRead

TapeUnit Read

Read 0x80 (car)

ing-read is true

returns TAPEUNITIRG

Sets Ext End of Transfer

returns -1 (TAPEUNITIRG)

Returns

Returns

0 1 2 3 4 5 6 7 8 9 10  
\$

143 144 145 146 147 148 149 150 151 152

1000  
v  
>, 03449 03449 V M C T D 424000 W A 03014 03449 J 01021 1 ..  
1000 " 21

2000  
^ THIS IS A TEST OF OVERLAPPED CONVEX OPERATION \*

2150  
A B C D E F G U

३०४७  
विवरण ग्रन्थालय

1082

2020 ✓ Message ✓

3000  
✓  
|

488

1(4) 46 8  
v ✓ R

25

A  
10/7

$$1 \oplus 1 \cap -6 = 1 \oplus \underset{\text{invert!}}{1}$$

$$1\oplus 35 - 6 = \underline{1\oplus 29}$$

V G ④④ - - B

$\text{R} \neq 0 \text{ or } 115^\circ$

Y 001241

7:30

? 2221

T c<sub>u</sub>c 68245 T 11

chain 'y'

Ch-1

OP-MOD-SIMBOL -

✓ *	✓ 1	overlap ch 1	*	1	
✓	✓ 2	overlap ch 2		2	
✓	✓ U	unit on 1 A	4		
✓	✓ F	.. .. 2BA	42		
✓	✓ Q	Ingoing 1 B	8		
	✓ *	Ingoing 2 B	84		DSTERISK
✓	✓ N	Outgoing B	4	1	
✓	✓ *	A	8	2	RM
✓	✓ S	Seek A		2	
✓	✓ J	A		2	1
✓	✓ A	Attention			
✓	✓ B				

Priority Request {

Op-BP PR, IOUWA PR, Injury PR, Outgoing PR  
Seek PR Attention PR

Instruction Priority Branches

OP-BP-BSCPR

OP-BI-ANCH-PR

Priority Alert in CCR

HT

$$J = 2 + 4 + 16 + 256 + \textcircled{2048}$$

$$B = 2 + 8 + 64 + 256 + 2048 + 4096$$

$$Y = 2 + 4 + 16 + 256 + \textcircled{1024}$$

$$128 \quad 32 + 128 + 256 + 2048$$

$$64 + 128 \quad 1 + 8 + 32$$

$$128 \quad 32 + 128 + 256 + 2048$$

Unit OTL overlap ~~start operation (in channel)~~  
~~wait for ext end of transfer in DQ overlay?~~  
followed by Status Sample

~~Read~~ Console WLR

Clear of Interlock Should also clear Read/Work

B - not overlapped

Branch I/O status - wants  
(call gate)

do overlap, check status

10 prob  
10 OK  
20

AC<sup>3</sup> 44<sup>10</sup>

91 44<sup>10</sup>

99 S/N B, but was

~~WLR~~ UDIM ?

~~WTR~~ SIB overlapped

Performance (only overlapped)

~~Even parity~~ write blank as C+A

Log Rewind (May be branch while overlapped)

Overl~~g~~

1 char per .016 ms

.016/.0045 ~~Work~~ - 3270 chars  
= 3.5 cycles

TBusy Device  
SetBusy  
TestBusy  
BusyPass  
Reset

Rewind = record #  
Span (2)

BSPC27

each 1024

$0.045 \times 10^{-6}$ , .0045 ms/cycle

BSP = 410 ms

8888

(for 2098)

use ~~222~~  $2 \times 1024 = 222$

10

WTR

$\approx 13$  ms

TR 10

222

Do Overlap

Status Sample

Do Output

Do Input

{  
Display  
WorkIMG  
Reset Indicator  
Stop Indicator  
GetStatus

Revert  $\rightarrow$  B-Reg

Storage wyp Check  $\rightarrow$  STAR Storage Latch

A Channel  $\rightarrow$  Set

STARMod  $\checkmark$  STAR

Assembly Channel  $\rightarrow$  Select

TAU Do Input OK

Do Input Read OK

Cooling

B-Reg

STAR

Storage wyp Latch

GetR2Status free

dynamic parameters, can  
Dynamic model, ui, us

Freq Spec

Get Feedback (www.wienerberger.com)

Shutter + 400

~~400~~

```
1: //-----
2: #ifndef UITAPEUNITH
3: #define UITAPEUNITH
4: //-----
5:
6: // Define constants returned by Read()
7:
8: #define TAPEUNITIRG      (-1)
9: #define TAPEUNITEOF      (-2)
10: #define TAPEUNITNOTREADY (-3)
11: #define TAPEUNITERROR    (-4)
12:
13: #define TAPE_IRG        0x80
14: #define TAPE_TM         0x0f
15:
16: class TTapeUnit : public TObject {
17:
18: protected:
19:
20:     int unit;                                // Unit number.
21:
22:     char filename[MAXPATH];                  // Path to tape file
23:     FILE *fd;                               // File Descriptor for I/O
24:     char tape_buffer;                      // Char read from file
25:
26:     int record_number;                     // Record number (0=bot)
27:
28:     bool loaded;                           // State flags
29:     bool fileprotect;
30:     bool ready;
31:     bool selected;
32:     bool tapeindicate;
33:     bool highdensity;
34:     bool bot;
35:     bool irg;
36:
37: private:
38:     int ReadNextChar();                   // Factored I/O call
39:
40:
41: public:
42:
43:     // Functions to make part of state visible to the outside
44:
45:     inline bool Selected() { return selected; }
46:     inline bool IsReady() { return ready; }
47:     inline bool IsFileProtected() { return fileprotect; }
48:     inline bool TapeIndicate() { return tapeindicate; }
49:     inline bool HighDensity() { return highdensity; }
50:     inline char *GetFileName() { return filename; }
51:     inline bool IsLoaded() { return loaded; }
52:     inline int GetRecordNumber() { return record_number; }
53:
54:     // Interface functions for the User Interface buttons
55:
56:     bool LoadRewind();
57:     bool Reset();
58:     bool Unload();
59:     bool Start();
60:     bool ChangeDensity();
61:     bool Mount(char *filename);
62:
63:     // Interface functions for the Tape Adapter Unit (TAU) to use
64:
65:     TTapeUnit(int u);                      // Constructor
66:     void Init(int u);                     // Initialization
```

```
67:           bool Select(bool b);                      // Select or De-select unit
68:           bool Rewind();                            // Skip and blank tape
69:           bool RewindUnload();                     // Mark end of record
70:           bool Backspace();                         // Write Tape Mark (EOF)
71:           bool Skip();                             // Space fwd 1 record
72:           void WriteIRG();                        // Write file character
73:           bool WriteTM();                          // Returns char or -value
74:           int Space();                           // Read()
75:           int Read();                            // Write(char c);
76:       };
77:   };
78:   bool Write(char c);
79: };
80:
81: #endif
```

```
1: //-----
2: #include <vcl.h>
3: #pragma hdrstop
4:
5: #include <dir.h>
6: #include <stdio.h>
7: #include <assert.h>
8: #include <errno.h>
9: #include "UITAPEUNIT.h"
10:
11: //-----
12: #pragma package(smart_init)
13:
14: #include "UI1410DEBUG.H"
15:
16: // Tape Unit Implementation.
17:
18: // Constructor
19:
20: TTapeUnit::TTapeUnit(int u) {
21:
22:     fd = NULL;                                // Let common init take over
23:     Init(u);
24: }
25:
26: // Initialization (not sure if anyone else will ever use this)
27:
28: void TTapeUnit::Init(int u) {
29:
30:     if(fd != NULL) {
31:         fclose(fd);
32:     }
33:
34:     unit = u;
35:     fd = NULL;
36:     irg = loaded = fileprotect = tapeindicate = ready = selected = bot = false;
37:     highdensity = true;
38:     filename[0] = '\0';
39:     record_number = 0;
40:     return;
41: }
42:
43: // Methods that interface to user interface buttons
44:
45: bool TTapeUnit::Reset() {
46:     ready = false;
47:     return(true);
48: }
49:
50: // Load the tape (file) (if not already loaded) and rewind.
51:
52: bool TTapeUnit::LoadRewind() {
53:
54:     if(ready) {                                // Inop if drive is ready
55:         return(false);
56:     }
57:
58:     if(fd != NULL) {                          // If loaded, just rewind
59:         if(fseek(fd,0L,SEEK_SET) < 0) {
60:             DEBUG("LoadRewind: fseek on failed on tape unit ",unit);
61:             DEBUG("LoadRewind: errno is ",errno);
62:             fclose(fd);
63:             fd = NULL;
64:             return(false);
65:         }
66:     }
}

```

✓ added code to  
write closing IRG

```
67:     else if(strlen(filename) == 0) {
68:         return(false);
69:     }
70:
71: // Open the file. First try RW. If that fails, try RO and set fileprot.
72:
73: if((fd = fopen(filename,"rb+")) == NULL) {          // Open RW. OK?
74:     if((fd = fopen(filename,"rb")) == NULL) {        // No. How about RO?
75:         DEBUG("LoadRewind: fopen failed on tape unit ",unit);
76:         DEBUG("LoadRewind: errno is ",errno);
77:         ready = loaded = fileprotect = bot = tapeindicate = irg = false;
78:         return(false);
79:     }
80:     else {                                         // RO OK
81:         fileprotect = true;
82:     }
83: }
84: else {                                         // RW OK
85:     fileprotect = false;
86: }
87:
88: irg = tapeindicate = false;
89: record_number = 0;
90: return(bot = loaded = true);
91: }
92:
93: // Unload the tape (file)
94:
95: bool TTapeUnit::Unload() {
96:
97:     if(ready || !loaded) {                         // If ready, ignore.
98:         return(false);
99:     }
100:
101:    if(fd != NULL) {
102:        fclose(fd);
103:        fd = NULL;
104:    }
105:    irg = loaded = fileprotect = tapeindicate = bot = false;
106:    record_number = 0;
107:    return(true);
108: }
109:
110: // Mount a tape on the drive (associate a file)
111:
112: bool TTapeUnit::Mount(char *fname) {
113:
114:     if(ready || loaded) {                         // If ready or already
115:         return(false);                           // loaded, ignore it.
116:     }
117:
118:     if(strlen(fname) == 0 || strlen(fname)+1 > sizeof(filename)) {
119:         return(false);
120:     }
121:     assert(fd == NULL);
122:     strcpy(filename,fname);
123:     irg = fileprotect = tapeindicate = bot = false;
124:     return(true);
125: }
126:
127: // Start Button
128:
129: bool TTapeUnit::Start() {
130:
131:     if(ready || !loaded) {                         // If ready or not loaded
132:         return(false);                           // can't help you!
```

```
133:     }
134:     assert(fd != NULL);
135:     return(ready = true);
136: }
137:     return(ready = true);
138: }
139:
140: bool TTapeUnit::ChangeDensity() {
141:     if(ready) {
142:         return(false);
143:     }
144:     highdensity = !highdensity;
145:     return(true);
146: }
147:
148: // Methods that interface with the Tape Adapter Unit (TAU)
149:
150: bool TTapeUnit::Select(bool b) {
151:     selected = b;
152:     return(true);
153: }
154:
155: // Rewind to beginning of tape (file)
156:
157: bool TTapeUnit::Rewind() {
158:
159:     if(!selected || !loaded || !ready) {           // Must be ready to go...
160:         DEBUG("TTapeUnit::Rewind: Unit not selected or not ready",unit);
161:         return(false);
162:     }
163:
164:     assert(fd != NULL);
165:
166:     if(fseek(fd,0L,SEEK_SET) < 0) {
167:         DEBUG("Rewind: fseek failed, unit ",unit);
168:         DEBUG("Rewind: errno is ",errno);
169:         fd = NULL;
170:         irg = loaded = ready = bot = false;
171:         return(false);
172:     }
173:
174:     irg = tapeindicate = false;
175:     record_number = 0;
176:     return(bot = true);
177: }
178:
179: // Rewind and unload the tape (close the file)
180:
181: bool TTapeUnit::RewindUnload() {
182:
183:     if(!Rewind()) {                                // If rewind fails,
184:         fclose(fd);                                // close the file anyway
185:         fd = NULL;
186:         return(false);
187:     }
188:
189:     if(fd != NULL) {
190:         fclose(fd);
191:         fd = NULL;
192:     }
193:
194:     irg = loaded = fileprotect = ready = tapeindicate = bot = false;
195:     record_number = 0;
196:     return(true);
197: }
198:
```

```
199: // Skip and blank tape, does nothing for now (until we have measured tape)
200:
201: bool TTapeUnit::Skip() {
202:
203:     if(!selected || !loaded || !ready) {
204:         return(false);
205:     }
206:     ++record_number;
207:     return(true);
208: }
209:
210: // Space forward. (d-character is "A" - not in my Principles of Operation!)
211: // Basically pretty easy: all we have to do is call read until we get a
212: // negative return code, which will happen at EOF or IRG or an error.
213: // We can let the Tape Adapter Unit figure out the status.
214:
215: int TTapeUnit::Space() {
216:
217:     int rc;
218:
219:     while((rc = Read()) >= 0) {
220:         // Do nothing.
221:     }
222:     ++record_number;
223:     return(rc);
224: }
225:
226: // Backspace. This one is a pain. To do it, we take two steps back, one
227: // forward. Repeatedly. Slow. Oh well....
228:
229: bool TTapeUnit::Backspace() {
230:
231:     if(!selected || !ready || !loaded) {
232:         DEBUG("TTapeUnit::Backspace: Unit not selected or not ready",unit);
233:         return(false);
234:     }
235:
236:     assert(fd != NULL);
237:
238:     if(bot) {
239:         record_number = 0;
240:         tapeindicate = true; // Backspace over BOT
241:         return(true);
242:     }
243:
244:     while(true) { // Start the dance...
245:
246:         // Seek back 2 characters
247:
248:         if(fseek(fd,-2L,SEEK_CUR) != 0) {
249:             DEBUG("Backspace: lseek failed on unit ",unit);
250:             DEBUG("Backspace: errno is ",errno);
251:             fclose(fd);
252:             fd = NULL;
253:             irg = loaded = fileprotect = ready = tapeindicate = false;
254:             return(false);
255:         }
256:
257:         // If beginning of file, done! (special case)
258:
259:         if(ftell(fd) == 0L) {
260:             irg = false;
261:             bot = true;
262:             record_number = 0;
263:             return(true);
264:         }
```

```
265:         // Read forward 1 character.  Quit on error or EOF
266:
267:
268:     if(fread(&tape_buffer,1,1,fd) != 1) {
269:         if(!feof(fd)) {
270:             DEBUG("Backspace: fread failed on unit ",unit);
271:             DEBUG("Backspace: errno is ",errno);
272:             fclose(fd);
273:             fd = NULL;
274:             irg = loaded = fileprotect = ready = tapeindicate = false;
275:             return(false);
276:         }
277:         DEBUG("Backspace: fread failed: unexpected eof on unit ",unit);
278:         return(false);
279:     }
280:
281:     // If we find an IRG bit on, we are done!
282:
283:     if(tape_buffer & TAPE_IRG) {
284:         tape_buffer ^= TAPE_IRG;
285:         irg = true;
286:         --record_number;
287:         return(true);
288:     }
289: }
290: }
291:
292: // Method to write a character. Note that by this time any cute stuff
293: // (like parity, changing wordmarks in to word separators, changing
294: // word separators into two word separators, etc. should have already been
295: // handled in the TAU
296:
297: bool TTapeUnit::Write(char c) {
298:
299:     if(irg) {
300:         c |= TAPE_IRG;
301:     }
302:
303:     if(!loaded || !ready || tapeindicate || !selected || fd == NULL) {
304:         DEBUG("TapeUnit::Write: Unit not ready or selected",unit);
305:         return(false);
306:     }
307:
308:     if(fileprotect) {
309:         DEBUG("TapeUnit::Write: Attempt to write when file protected, unit ",
310:               unit);
311:         return(false);
312:     }
313:
314:     if(fwrite(&c,1,1,fd) != 1) {
315:         DEBUG("TapeUnit::Write: File I/O error writing on unit ",unit);
316:         DEBUG("TapeUnit::Write: errno is ",errno);
317:         fclose(fd);
318:         fd = NULL;
319:         tapeindicate = true;
320:         irg = ready = loaded = fileprotect = bot = false;
321:         return(false);
322:     }
323:
324:     irg = false;
325:     return(true);
326: }
327:
328: // Mark end of record. Called at the end of a transfer by the TAU.
329: // All this does is set the IRG flag for the start of the next record.
330:
```

```
331: void TTapeUnit::WriteIRG() {
332:     irg = true;
333:     ++record_number;
334:     return;
335: }
336:
337: // Write tape mark. Just calls write to do the dirty work...
338: // Note that tape marks are *always* even parity
339:
340: bool TTapeUnit::WriteTM() {
341:     ++record_number;
342:     return(Write(TAPE_TM | TAPE_IRG));
343: }
344:
345: // Read a character. Has to handle one interesting case. The tapes
346: // have an extra Tape Mark for each tape mark, so we have to skip that
347: // extra one. (The 2nd Tape Mark does NOT have IRG set).
348:
349: int TTapeUnit::Read() {
350:
351:     int rc;
352:
353:     if(!loaded || !ready || !selected || tapeindicate || fd == NULL) {
354:         DEBUG("TapeUnit::Read: Unit not ready or selected: ",unit);
355:         return(TAPEUNITNOTREADY);
356:     }
357:
358:     // Read a character, unless the last read resulted in an IRG, in
359:     // which case it is already in the buffer.
360:     // Interesting statuses are negative, just bubble them on up.
361:
362:     if(!irg) {
363:         if((rc = ReadNextChar()) < 0) {
364:             return(rc);
365:         }
366:         tape_buffer = (char) rc;
367:     }
368:
369:     // If at an irg and the character read is a TM, skip bogus next
370:     // char, and return EOF indication.
371:
372:     if((tape_buffer & 0x3f) == TAPE_TM && (tape_buffer & TAPE_IRG) != 0) {
373:         if((rc = ReadNextChar()) < 0) {
374:             return(rc);
375:         }
376:         ++record_number;
377:         return(TAPEUNITEOF);
378:     }
379:
380:     // If we are not at bot or an IRG, strip the IRG bit from the char.
381:     // We will then just return that char.
382:
383:     if(bot || irg) {
384:         tape_buffer &= (~TAPE_IRG);
385:         bot = irg = false;
386:     }
387:
388:     // If we hit the end of the record, then set irg now, and return such
389:
390:     if(tape_buffer & TAPE_IRG) {
391:         irg = true;
392:         ++record_number;
393:         return(TAPEUNITIRG);
394:     }
395:
396:     // Aw shucks, just return the blinkin' character already!
```

```
397:         return(tape_buffer);
398:     }
399: }
400:
401: // Utility method to read a character from the file, and handle a few odds
402: // and ends. Keeps us from having to do it all more than once...
403:
404: int TTapeUnit::ReadNextChar() {
405:
406:     unsigned char c;
407:
408:     if(fread(&c,1,1,fd) != 1) {
409:         if(!feof(fd)) {
410:             DEBUG("TapeUnit::ReadNextChar: I/O Error Reading File, unit ",unit);
411:             DEBUG("TapeUnit::ReadNextChar: errno is ",errno);
412:             irg = loaded = fileprotect = ready = false;
413:             tapeindicate = true;
414:             return(TAPEUNITERROR);
415:         }
416:         tapeindicate = true;
417:         return(TAPEUNITEOF);
418:     }
419:
420:     return(c);
421: }
422:
```

```
1: //-----
2: #ifndef UITAPETAUH
3: #define UITAPETAUH
4: //-----
5:
6: #define TAPE_IO_DEVICE 20
7:
8: #define UNIT_BACKSPACE 50
9: #define UNIT_SKIP 53
10: #define UNIT_REWIND 41
11: #define UNIT_REWIND_UNLOAD 20
12: #define UNIT_WTM 36
13: #define UNIT_SPACE 49
14:
15: // 1410 Tape Adapter Unit
16:
17: class TTapeTAU : public T1410IODevice {
18:
19: protected:
20:
21:     TTapeUnit *Unit[10]; // Units 0 thru 9
22:     TTapeUnit *TapeUnit; // Current selected unit
23:
24:     int tapestatus; // Channel status
25:     BCD ch_char; // Channel character
26:
27:     char tape_char; // Tape Unit character
28:     int tape_read_char; // Before checking status
29:
30:     long chars_transferred; // Storage characters
31:     char *tape_parity_table;
32:
33:
34: public:
35:
36:     // Implement the I/O device interface standard
37:
38:     TTapeTAU(int devicenum, T1410Channel *Channel);
39:     virtual int Select();
40:     virtual void DoOutput();
41:     virtual void DoInput();
42:     virtual int StatusSample();
43:     virtual void DoUnitControl(BCD opmod);
44:
45:     // State and status methods
46:
47:     TTapeUnit *GetUnit(int unit);
48:
49: private:
50:
51:     // Internal methods.
52:
53:     bool DoOutputWrite(char c);
54:     int DoInputRead();
55: };
56:
57:
58: #endif
```

```
1: //-----
2: #include <vcl.h>
3: #pragma hdrstop
4:
5: #include <dir.h>
6: #include <stdio.h>
7: #include <assert.h>
8: #include <errno.h>
9: #include "UBCD.H"
10: #include "UI1410CPUUT.H"
11: #include "UI1410CHANNEL.H"
12: #include "UITAPEUNIT.h"
13: #include "UITAPETAU.h"
14: #include "UI729TAPE.H"
15:
16:
17: //-----
18: #pragma package(smart_init)
19:
20: // 1410 Tape Adapter Unit Implementation.  Follows I/O Device Interface.
21:
22: // Constructor.  Creates tape drives!
23: // NOTE: This is expected to be called with the EVEN PARITY designation
24: // for the device number ("U").  This constructor automatically adds the
25: // ODD PARITY designation ("B")!
26:
27: TTapeTAU::TTapeTAU(int devicenum,T1410Channel *Channel) :
28:     T1410IODevice(devicenum,Channel) {
29:
30:     static char parity_table[128] = {
31:         0,1,1,0,1,0,0,1,1,1,0,0,1,0,1,0,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,
32:         1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,
33:         1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,
34:         0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1,0,1,0,0,1,1,0,
35:     };
36:
37:     int i;
38:
39:     for(i=0; i < 10; ++i) {
40:         Unit[i] = new TTapeUnit(i);
41:     }
42:
43:     TapeUnit = NULL;
44:     tapestatus = 0;
45:     chars_transferred = 0;
46:     tape_parity_table = parity_table;
47:
48:     // Create the odd parity device now...
49:
50:     Channel -> AddIODevice(this,BCD(BCD::BCDConvert('B')).To6Bit());
51: }
52:
53: // Select.  If some other unit is selected, it gets deselected.
54: // The unit comes from the CPU / Channel
55:
56: int TTapeTAU::Select() {
57:
58:     int u;
59:
60:     if(TapeUnit != NULL) {
61:         TapeUnit -> Select(false);
62:         TapeUnit = NULL;
63:     }
64:
65:     // Find the tape unit.  Give up if none there.  Otherwise, select it.
66:
```

```
67:     u = Channel -> GetUnitNumber();
68:     if((TapeUnit = GetUnit(u)) == NULL) {
69:         return(tapestatus = IOCHNOTREADY);
70:     }
71:     TapeUnit -> Select(true);
72:
73:     // Set 0 length record so far.
74:
75:     chars_transferred = 0;
76:
77:     // If writing, and tape is write protected, set not ready and condition.
78:     // Not sure if that is write - the diagnostics will probably figure it out
79:     // for me... 8-
80:
81:     if(Channel -> ChWrite -> State() && TapeUnit -> IsFileProtected()) {
82:         return(tapestatus |= (IOCHNOTREADY | IOCHCONDITION));
83:     }
84:
85:     return(tapestatus = 0);
86: }
87:
88: // Return a pointer to a given tape drive, NULL if invalid.
89:
90: TTapeUnit *TTapeTAU::GetUnit(int u) {
91:
92:     if(u < -0 || u > 9) {
93:         return(NULL);
94:     }
95:     return(Unit[u]);
96: }
97:
98: // DoOutput: Accept an output character from the channel.
99:
100: void TTapeTAU::DoOutput() {
101:
102:     // If no (or invalid) unit selected, just return not ready.
103:
104:     if(TapeUnit == NULL) {
105:         tapesstatus |= IOCHNOTREADY;
106:         Channel -> ExtEndofTransfer = true;
107:         return;
108:     }
109:
110:     // If file protected, return not ready as well. (Not sure if this is
111:     // correct, but hopefully the diagnostics will test it)
112:
113:     if(TapeUnit -> IsFileProtected()) {
114:         tapesstatus |= IOCHNOTREADY;
115:         Channel -> ExtEndofTransfer = true;
116:         return;
117:     }
118:
119:     // Get the character from the channel. Strip down to 6 bits for tape.
120:
121:     ch_char = Channel -> ChR2 -> Get();
122:     tape_char = ch_char.To6Bit();
123:
124:     // If we are in load mode, and the character either has a word mark or is
125:     // itself a word separator, write out a word separator character to tape.
126:
127:     if((Channel -> LoadMode && ch_char.TestWM()) ||
128:         tape_char == (BCD_WS & 0x3f)) {
129:         if(!DoOutputWrite(BCD_WS)) {
130:             tapesstatus |= IOCHCONDITION;
131:             Channel -> ExtEndofTransfer = true;
132:             return;
```

```
133:         }
134:     }
135:
136: // If the incoming character has invalid parity, flag a datacheck. But
137: // we will still write what we can, in valid parity, to tape.
138:
139: if(!ch_char.CheckParity()) {
140:     tapestatus |= IOCHDATACHECK;
141: }
142:
143: // Finally, write out the character.
144:
145: if(!DoOutputWrite(tape_char)) {
146:     tapestatus |= IOCHCONDITION;
147:     Channel -> ExtEndofTransfer = true;
148:     return;
149: }
150: both
151: ++chars_transferred;
152: }
153:
154: // Method to tell channel when we have data.
155:
156: void TTapeTAU::DoInput() {
157:
158:     bool wm = false;
159:     BCD b;
160:
161:     if(TapeUnit == NULL) {
162:         tapestatus |= IOCHNOTREADY;
163:         Channel -> ExtEndofTransfer = true;
164:         return;
165:     }
166:
167:     // If tape indicate is on, say we are not ready too.
168:
169:     if(TapeUnit -> TapeIndicate()) {
170:         tapestatus |= IOCHNOTREADY;
171:         Channel -> ExtEndofTransfer = true;
172:         return;
173:     }
174:
175:     // Get a character from the tape unit. If an interesting status, just
176:     // return. The utility method will have already set the right status.
177:     // (That is why the utility method exists!)
178:
179:     tape_read_char = DoInputRead();
180:     if(tape_read_char < 0) {
181:         return;
182:     }
183:
184:     ++chars_transferred;
185:
186:     // If we got a tape mark, tell the channel to store it. The TAU always
187:     // returns tape marks in EVEN PARITY. The channel will store it as is
188:     // (and flag a machine check) unless asterisk insert is on!
189:
190:     if((tapestatus & IOCHCONDITION) &&
191:         tape_read_char == BCD_TM) {
192:         Channel -> ChannelStrobe(BCD_TM);
193:         return;
194:     }
195:
196:     // If we are reading in load mode, and we got a word separator, read the
197:     // next character. Then we will either set a WM on it, or, if it too is
198:     // a word separator, just store the word separator. If we have a problem
```

both { Channel → OutputRequest = true;  
Channel → CycleRequest = true;

```
199: // reading the next char, just return -- the read routine will have set
200: // status -- that is why it is there! 8-
201:
202: if(Channel -> LoadMode && (tape_read_char & 0x3f) == (BCD_WS & 0x3f)) {
203:     wm = true;
204:     tape_read_char = DoInputRead();
205:     if(tape_read_char < 0) {
206:         return;
207:     }
208:     if((tape_read_char & 0x3f) == BCD_WS) {
209:         wm = false;
210:     }
211: }
212:
213: b = BCD(tape_read_char);
214:
215: // Now, the character we just got is in the correct TAPE parity. If we
216: // are reading tape in EVEN parity, we have to FLIP THE CHECK BIT!
217:
218: if((Channel -> ChUnitType -> Get().ToInt() & 2) == 0) { // B as 2 bit - ODD
219:     // NO 2 bit -- EVEN!
220:     b.ComplementCheck();
221: }
222:
223: // If we set the wm flag earlier, turn it on (and flip the check bit)
224:
225: if(wm) {
226:     b.SetWM();
227:     b.ComplementCheck();
228: }
229:
230: // Finally, tell the channel we have something to eat.
231:
232: Channel -> ChannelStrobe(b);
233: }
234:
235: // Method for unit control. Mostly, just passes it off to the tape unit.
236: // Also sets ExtEndofTransfer, indicating we are done. (Will probably have
237: // to work in delays someday. Oh well)
238:
239: void TTapeTAU::DoUnitControl(BCD opmod) {
240:
241:     int d;
242:     int rc;
243:
244:     d = opmod.To6Bit();
245:
246:     if(TapeUnit == NULL) {
247:         tapestatus |= IOCHNOTREADY;
248:         Channel -> ExtEndofTransfer = true;
249:         return;
250:     }
251:
252:     switch(d) {
253:
254:     case UNIT_BACKSPACE:
255:         if(!TapeUnit -> Backspace()) {
256:             tapestatus |= IOCHNOTREADY;
257:         }
258:         if(TapeUnit -> TapeIndicate()) {
259:             tapestatus |= IOCHCONDITION;
260:         }
261:         break;
262:
263:     case UNIT_SKIP:
264:         if(!TapeUnit -> Skip()) {
```

```
265:             tapestatus |= IOCHNOTREADY;
266:         }
267:         break;
268:
269:     case UNIT_REWIND:
270:         if(!TapeUnit -> Rewind()) {
271:             tapestatus |= IOCHNOTREADY;
272:         }
273:         break;
274:
275:     case UNIT_REWIND_UNLOAD:
276:         if(!TapeUnit -> Unload()) {
277:             tapestatus |= IOCHNOTREADY;
278:         }
279:         break;
280:
281:     case UNIT_WTM:
282:         if(!TapeUnit -> WriteTM()) {
283:             tapestatus |= IOCHNOTREADY;
284:         }
285:         // Write TM in odd parity sets Data Check
286:         if(Channel -> ChUnitType -> Get().ToInt() & 2) {
287:             tapestatus |= IOCHDATACHECK;
288:         }
289:         break;
290:
291:     case UNIT_SPACE:
292:         rc = TapeUnit -> Space();
293:         switch(rc) {
294:             case TAPEUNITEOF:
295:                 tapestatus |= IOCHCONDITION;
296:                 break;
297:             case TAPEUNITNOTREADY:
298:                 tapestatus |= IOCHNOTREADY;
299:                 break;
300:             case TAPEUNITERROR:
301:                 tapestatus |= (IOCHNOTREADY | IOCHCONDITION);
302:                 break;
303:             default:
304:                 break;
305:         }
306:         break;
307:
308:     default:
309:         tapestatus |= IOCHNOTREADY;
310:         break;
311:     }
312:
313:     FI729 -> Display();
314:     Channel -> ExtEndofTransfer = true;
315:     return;
316: }
317:
318: // Private utility methods...
319:
320: // Utility method to write a character to the drive and collect status.
321:
322: bool TTapeTAU::DoOutputWrite(char c) {
323:
324:     int wanted_parity;
325:
326:     if(TapeUnit == NULL) {
327:         tapestatus |= IOCHNOTREADY;
328:         Channel -> ExtEndofTransfer = true;
329:         return(false);
330:     }
```

```
331:     c &= 0x3f;                                // Bye bye WM & Check Bit.
332:
333:
334: // Check the unit type. B, which has a "2" bit, means odd.
335:
336: wanted_parity = ((Channel -> ChUnitType -> Get().ToInt() & 2) != 0); // 1 Odd
337:
338: if(tape_parity_table[c] != wanted_parity) {
339:     c ^= BITC;
340: }
341:
342: // Finally, tell the drive to write the character.
343:
344: return(TapeUnit -> Write(c));
345: }
346:
347: // Utility method to read a character from a tape drive. Sets appropriate
348: // status if the drive returns something other than an ordinary character.
349:
350: int TTapeTAU::DoInputRead() {
351:
352:     int c;
353:     bool wanted_parity;
354:
355:     c = TapeUnit -> Read();
356:
357:     if(c < 0) {
358:         Channel -> ExtEndofTransfer = true;
359:         if(c == TAPEUNITNOTREADY || c == TAPEUNITERROR) {
360:             tapestatus |= IOCHNOTREADY;
361:             return(c);
362:         }
363:         else if(c == TAPEUNITEOF) {
364:             tapestatus |= IOCHCONDITION;
365:             return(0);
366:         }
367:         else {
368:             return(c);
369:         }
370:     }
371:
372: // Check what parity we want. We want odd parity if the device in the
373: // instruction was "B" (has a 2 bit).
374:
375: wanted_parity = ((Channel -> ChUnitType -> Get().ToInt() & 2) != 0);
376:
377: assert(c >= 0 && c < 0x80);
378: if(tape_parity_table[c] != wanted_parity) {
379:     tapestatus |= IOCHDATACHECK;
380: }
381:
382: return(c);
383: }
384:
385: // Return status at end of operation
386:
387: int TTapeTAU::StatusSample() {
388:     if(TapeUnit != NULL) {
389:         FI729 -> Display();
390:     }
391:     return(Channel -> GetStatus() | tapestatus);
392: }
```

*(Handwritten notes and annotations)*

*C = BCD-TM*

```

1: //-
2: #ifndef UI1410CHANNELH
3: #define UI1410CHANNELH
4: //-
5:
6: // This class defines what is in an I/O Channel
7:
8: #define IOCHNOTREADY 1
9: #define IOCHBUSY 2
10: #define IOCHDATACHECK 4
11: #define IOCHCONDITION 8
12: #define IOCHNOTTRANSFER 16
13: #define IOCHWLRECORD 32
14:
15: #define IOLAMPNOTREADY 0
16: #define IOLAMPBUSY 1
17: #define IOLAMPDATACHECK 2
18: #define IOLAMPCONDITION 3
19: #define IOLAMPNOTTRANSFER 4
20: #define IOLAMPWLRECORD 5
21:
22: // Forward declaration of I/O Device class, which T1410Channel uses.
23:
24: class T1410IODevice;
25:
26: class T1410Channel : public TDisplayObject {
27:
28: public:
29:
30:     void Reset();
31:
32:     // Functions inherited from abstract base class classes now need definition
33:
34:     void OnComputerReset();
35:     void OnProgramReset();
36:     void Display();
37:     void LampTest(bool b);
38:
39: private:
40:
41:     // Channel information
42:
43:     int ChStatus; // Channel status (see defines)
44:     TLabel *ChStatusDisplay[6]; // Channel status lights
45:     bool R1Status, R2Status; // State of R1, R2. True if full
46:     class T1410IODevice *Devices[64]; // Pointer to devices
47:
48: public:
49:
50:     TRegister *ChOp;
51:     TRegister *ChUnitType;
52:     TRegister *ChUnitNumber;
53:     TRegister *ChR1, *ChR2; // Address Register *ChAddr;
54:
55:     TDisplayLatch *ChInterlock; // Top Indicator of Methods
56:     TDisplayLatch *ChRBCLock;
57:     TDisplayLatch *ChRead;
58:     TDisplayLatch *ChWrite;
59:     TDisplayLatch *ChOverlap;
60:     TDisplayLatch *ChNotOverlap;
61:
62:     bool MoveMode, LoadMode; // Mode: Without/With WM
63:     bool IntEndofTransfer, ExtEndofTransfer; // Transfer end flags
64:     bool CycleRequired; // Request for input transfer
65:     bool InputRequest; // Not in real machine - for input
66:                           // co-routine.

```

```

67:     bool LastInputCycle;           // Indicate last cycle of input
68:     bool EndofRecord;            // Indicate record has ended
69:     T1410IODevice *CurrentDevice; // Ptr to device doing transfer
70:
71:     enum TapeDensity {
72:         DENSITY_200_556 = 0, DENSITY_200_800 = 1, DENSITY_556_800 = 2
73:     } TapeDensity;
74:
75:
76:     // Methods
77:
78:     T1410Channel(
79:         TLabel *LampInterlock,      TAddressRegister *Addr, // Constructor
80:         TLabel *LampRBCInterlock,
81:         TLabel *LampRead,
82:         TLabel *LampWWrite,
83:         TLabel *LampOverlap,
84:         TLabel *LampNotOverlap,
85:         TLabel *LampNotRead,
86:         TLabel *LampBusy,
87:         TLabel *LampDataCheck,
88:         TLabel *LampCondition,
89:         TLabel *LampWLRecord,
90:         TLabel *LampNoTransfer
91:     );
92:
93:     // The following routine is called from T1410IODevice constructors
94:     // to add devices into the Channel's device table.
95:
96:     void AddIODevice(T1410IODevice *iodevice, int devicenumber);
97:
98:     // Methods that do most of the Channel's real work.
99:
100:    inline int SetStatus(int i) { return ChStatus = i; }
101:    inline int GetStatus() { return ChStatus; }
102:    inline int GetDeviceNumber() {
103:        return ChUnitType -> Get().ToInt() & 0x3f;
104:    }
105:    T1410IODevice *SetCurrentDevice() {
106:        CurrentDevice = Devices[GetDeviceNumber()];
107:    }
108:    T1410IODevice *GetCurrentDevice() { return CurrentDevice; }
109:    inline int GetUnitNumber() {
110:        return ChUnitNumber -> Get().ToAscii() - '0';
111:    }
112:
113:    void DoOutput(TAddressRegister *addr);           // Channel to Device
114:    void DoInput(TAddressRegister *addr);             // Device to Channel - transfer
115:    bool ChannelStrobe(BCD ch);                     // Device to Channel - Request
116:    void DoUnitControl(BCD opmod);                  // Unit control
117:
118:    // Channel register methods
119:
120:
121:    inline bool GetR1Status() { return R1Status; };
122:    inline bool GetR2Status() { return R2Status; };
123:    inline void ResetR1() { R1Status = false; };
124:    inline void ResetR2() { R2Status = false; };
125:    BCD SetR1(BCD b);
126:    BCD SetR2(BCD b);
127:    BCD MoveR1R2();
128: };
129:
130: // Class declaration for IO Devices. This is an *abstract* class,
131: // which is used to derive a class for each kind of IO device (console,
132: // reader, punch, tape, disk, etc. The Unit Control function is the

```

```
133: // only one that is not pure virtual. We provide a default for it because
134: // most devices don't implement unit control
135:
136: class T1410IODevice : public TObject {
137:
138: protected:
139:     T1410Channel *Channel; // Channel device is attached to
140:
141: public:
142:     T1410IODevice(int devicenum,T1410Channel *Channel); // Constructor
143:     virtual int Select() = 0; // Start an operation
144:     virtual int StatusSample() = 0; // Return device status
145:     virtual void DoOutput() = 0; // Channel -> Device
146:     virtual void DoInput() = 0; // Device -> Channel
147:     virtual void DoUnitControl(BCD opmod); // Unit Control
148: };
149:
150: #endif
```

```
1: //-----
2: #include <vcl.h>
3: #pragma hdrstop
4:
5: #include "UBCD.H"
6: #include "UI1410CPUH.H"
7: #include "UI1410CHANNEL.h"
8:
9: //-----
10: #pragma package(smart_init)
11:
12: #include <assert.h>
13: #include <stdio.h>
14:
15: #include "UI1410DEBUG.H"
16: #include "UI1410INST.H"
17: #include "UI1415IO.H"
18: #include "UI1415CE.H"
19:
20: // Implementation of I/O Channel Class
21:
22: // Constructor. Initializes state
23:
24: T1410Channel::T1410Channel( TAddressRegister *Addr,
25:                             TLabel *LampInterlock,
26:                             TLabel *LampRBCInterlock,
27:                             TLabel *LampRead,
28:                             TLabel *LampWrite,
29:                             TLabel *LampOverlap,
30:                             TLabel *LampNotOverlap,
31:                             TLabel *LampNotReady,
32:                             TLabel *LampBusy,
33:                             TLabel *LampDataCheck,
34:                             TLabel *LampCondition,
35:                             TLabel *LampWLRecord,
36:                             TLabel *LampNoTransfer ) {
37:
38:     int i;
39:
40:     ChStatus = 0;
41:     TapeDensity = DENSITY_200_556;
42:     R1Status = R2Status = false;
43:
44:     ChStatusDisplay[IO_LAMPNOTREADY] = LampNotReady;
45:     ChStatusDisplay[IO_LAMPBUSY] = LampBusy;
46:     ChStatusDisplay[IO_LAMPDATACHECK] = LampDataCheck;
47:     ChStatusDisplay[IO_LAMPCONDITION] = LampCondition;
48:     ChStatusDisplay[IO_LAMPNOTTRANSFER] = LampNoTransfer;
49:     ChStatusDisplay[IO_LAMPWLRECORD] = LampWLRecord;
50:
51:     // Generally, the channel latches are *not* reset by Program Reset
52:
53:     ChInterlock = new TDisplayLatch(LampInterlock, false);
54:     ChrBCInterlock = new TDisplayLatch(LampRBCInterlock, false);
55:     ChRead = new TDisplayLatch(LampRead, false);
56:     ChWrite = new TDisplayLatch(LampWrite, false);
57:     ChOverlap = new TDisplayLatch(LampOverlap, false);
58:     ChNotOverlap = new TDisplayLatch(LampNotOverlap, false);
59:
60:     // Generally, the channel registers are *not* reset by Program Reset
61:
62:     ChOp = new TRegister(false);
63:     ChUnitType = new TRegister(false);
64:     ChUnitNumber = new TRegister(false);
65:     Chr1 = new TRegister(false);
66:     Chr2 = new TRegister(false);
```

*ChAddr = Addr;*

```
67: // Clear the device table
68: for(i=0; i < 64; ++i) {
69:     Devices[i] = NULL;
70: }
71: CurrentDevice = NULL;
72: MoveMode = LoadMode = false;
73: ExtEndofTransfer = IntEndofTransfer = false;
74: CycleRequired = false;
75: InputRequest = false;
76: LastInputCycle = false;
77: EndofRecord = false;
78: }
79: }
80: }
81: }
82: }
83: // Channel Register methods
84: 
85: BCD T1410Channel::SetR1(BCD b) {
86:     ChR1 -> Set(b);
87:     R1Status = true;
88:     return(b);
89: }
90: }
91: BCD T1410Channel::SetR2(BCD b) {
92:     ChR2 -> Set(b);
93:     R2Status = true;
94:     return(b);
95: }
96: }
97: // Move data from register 1 to register 2
98: // Clearing register 1 in the process.
99: 
100: BCD T1410Channel::MoveR1R2() {
101:     ChR2 -> Set(ChR1 -> Get());
102:     R1Status = false;
103:     R2Status = true;
104:     return(ChR2 -> Get());
105: }
106: }
107: // Method to add a new device to the channel's device table.
108: // Typically this is called from the T1410IODevice base class constructor
109: 
110: void T1410Channel::AddIODevice(T1410IODevice *iodevice, int devicenumber) {
111:     assert(Devices[devicenumber] == NULL);
112:     Devices[devicenumber] = iodevice;
113: }
114: }
115: // Channel is reset during ComputerReset
116: 
117: void T1410Channel::OnComputerReset()
118: {
119:     Reset();
120: }
121: 
122: void T1410Channel::Reset() {
123:     ChStatus = 0;
124:     R1Status = R2Status = false;
125:     MoveMode = LoadMode = false;
126:     ExtEndofTransfer = IntEndofTransfer = false;
127:     CycleRequired = false;
128:     InputRequest = false;
129:     LastInputCycle = false;
130:     EndofRecord = false;
131:     ChInterlock -> Reset();
132:     ChrBCInterlock -> Reset();
```

```
133:     ChRead -> Reset();
134:     ChWrite -> Reset();
135:     ChOverlap -> Reset();
136:     ChNotOverlap -> Reset();
137: }
138:
139: // Channel is not reset during Program Reset
140:
141: void T1410Channel::OnProgramReset()
142: {
143:     // Channel not affected by Program Reset
144: }
145:
146: // Display Routine.
147:
148: void T1410Channel::Display() {
149:
150:     int i;
151:
152:     ChStatusDisplay[IOLAMPNOTREADY] -> Enabled =
153:         ((ChStatus & IOCHNOTREADY) != 0);
154:     ChStatusDisplay[IOLAMPBUSY] -> Enabled =
155:         ((ChStatus & IOCHBUSY) != 0);
156:     ChStatusDisplay[IOLAMPDATACHECK] -> Enabled =
157:         ((ChStatus & IOCHDATACHECK) != 0);
158:     ChStatusDisplay[IOLAMPCONDITION] -> Enabled =
159:         ((ChStatus & IOCHCONDITION) != 0);
160:     ChStatusDisplay[IOLAMPWLRECORD] -> Enabled =
161:         ((ChStatus & IOCHWLRECORD) != 0);
162:     ChStatusDisplay[IOLAMPNOTTRANSFER] -> Enabled =
163:         ((ChStatus & IOCHNOTTRANSFER) != 0);
164:
165:     for(i=0; i <= 5; ++i) {
166:         ChStatusDisplay[i] -> Repaint();
167:     }
168:
169:     // Although in most instances the following would be redundant,
170:     // because these objects are also on the CPU display list, we include
171:     // them here in case we want to display a channel separately.
172:
173:     ChInterlock -> Display();
174:     ChrBCInterlock -> Display();
175:     ChRead -> Display();
176:     ChWrite -> Display();
177:     ChOverlap -> Display();
178:     ChNotOverlap -> Display();
179: }
180:
181: // Channel Lamp Test
182:
183: void T1410Channel::LampTest(bool b)
184: {
185:     int i;
186:
187:     // Note, we don't have to do anything to the TDisplayLatch objects in
188:     // the channel for lamp test. They will take care of themselves on a
189:     // lamp test.
190:
191:     if(!b) {
192:         for(i=0; i <= 5; ++i) {
193:             ChStatusDisplay[i] -> Enabled = true;
194:             ChStatusDisplay[i] -> Repaint();
195:         }
196:     }
197:     else {
198:         Display();
```

```
199:     }
200: }
201:
202: // Channel Output to Device
203: // Conditions on Entry: B data register (B_REG) contains character to output
204: // Channel Register 1 (E1 or F1) should be empty.
205:
206: void T1410Channel::DoOutput(TAddressRegister *addr) {
207:
208:     BCD tempbcd;
209:
210:     CPU -> CycleRing -> Set(this == CPU -> Channel[CHANNEL1] ? CYCLE_E : CYCLE_F);
211:
212:     // Read out the next character. Also, check for a storage wrap, which will
213:     // end the transfer after this character.
214:
215:     *(CPU -> STAR) = *addr;
216:     CPU -> Readout();
217:     CPU -> StorageWrapCheck(+1);
218:     tempbcd = CPU -> B_Reg -> Get();
219:
220:     // If we have a GMWM, then we are done transferring data from memory
221:     // Otherwise, Clear the WM from the input data if in Move mode, and
222:     // then (regardless of mode) transfer the data to Channel register R1
223:     // Also, if we actually have read a character out, check for storage
224:     // wrap, which also ends the transfer.
225:
226:     if(tempbcd.TestGMWM()) {
227:         IntEndofTransfer = true;
228:         EndofRecord = true;
229:     }
230:     else {
231:         if(MoveMode) {
232:             tempbcd.ClearWM();
233:             tempbcd.SetOddParity();
234:         }
235:         SetR1(tempbcd);
236:         if(CPU -> StorageWrapLatch) {
237:             IntEndofTransfer = true;
238:             EndofRecord = true;
239:         }
240:     }
241:
242:     // Now, if we have anything for the device, send it. We continue
243:     // doing this for up to two characters, in case R1 and R2 are both
244:     // full. We have to do it this way so that if, at the end, we set
245:     // ExtEndofTransfer, all of the data will have been sent.
246:
247:     while(GetR1Status() || GetR2Status()) {
248:         if(GetR1Status() & !GetR2Status()) {
249:             MoveR1R2();
250:         }
251:         CurrentDevice -> DoOutput();
252:         ResetR2();
253:     }
254:
255:     // Advance the appropriate address register to the next location.
256:     // (Storage wrap was detected earlier, as necessary!).
257:
258:     addr -> Set(CPU -> STARMOD(+1));
259:
260:     // Now that the data is all sent, if we are at IntEndofTransfer (either
261:     // From a GMWM or from storage wrap), also set ExtEndofTransfer so that
262:     // we quit. Note that the device may have already set ExtEndofTransfer!
263:
264:     if(IntEndofTransfer) {
```

```

265:     ExtEndofTransfer = true;           CurrentDevice → DISCONNECTED;
266: }
267:
268: }
269:
270: // Channel Input Processing (shared overlap/not overlap code)
271:
272: void T1410Channel::DoInput(TAddressRegister *addr) {
273:
274:     CycleRequired = false;           // Reset cycle required
275:     InputRequest = false;          // Reset co-routine flag
276:
277:     if(!IntEndofTransfer) {         // Skip this if wrapped!
278:         *(CPU -> STAR) = *addr;    // Copy memory address
279:         CPU -> Readout();        // Get existing memory
280:         CPU -> AChannel -> Select( // Gate A channel approp.
281:             this == CPU -> Channel[CHANNEL1] ?
282:                 TACchannel::A_Channel_E : TACchannel::A_Channel_F );
283:     }
284:
285:     // If no data from device, the end is near...
286:
287:     if(!(GetR1Status() || GetR2Status())) {
288:         LastInputCycle = true;
289:     }
290:
291:     // If B GMWM, we are typically done storing data (unless this
292:     // is read to end of core, which ignores GMWM). Unless, of course,
293:     // we have already hit end of record.
294:
295:     // Note that we can only check the op mod if we are NOT overlapped
296:     // (Note that the "to end of core" mods are all not overlapped)
297:
298:     if(!EndofRecord && CPU -> B_Reg -> Get().TestGMWM()) {
299:         if(ChOverlap -> State() ||           ← Copy of ① (bump past GMWM)
300:             (CPU -> Op_Mod_Reg -> Get().To6Bit()) != OP_MOD_SYMBOL_DOLLAR) {
301:             EndofRecord = true;           ←
302:         }
303:     }
304:
305:     // If no more input, or we hit GMWM,
306:     // set internal end of transfer, but keep on accepting characters
307:     // until External end of transfer.
308:
309:     if(LastInputCycle || EndofRecord) {
310:         IntEndofTransfer = true;
311:         if(ExtEndofTransfer) {
312:             return;                   ←
313:         }
314:     }
315:
316:     // If we get here, we are either ending, or we are still storing data
317:
318:     if(!LastInputCycle && !IntEndofTransfer) {      // Still storing input?
319:
320:         // If parity is no good, set data check...
321:
322:         if(!CPU -> AChannel -> Select().CheckParity()) {
323:             SetStatus(GetStatus() | IOCHDATACHECK);
324:
325:             // If asterisk insert, store an asterisk!
326:
327:             if(FI1415CE -> AsteriskInsert -> Checked) {
328:                 ChR2 -> Set(BCD_ASTERISK);
329:                 CPU -> AChannel -> Select(
330:                     this == CPU -> Channel[CHANNEL1] ?

```

```

331:             }
332:         }
333:     } // End initial parity check
334:
335:     // Store the data (perhaps with a parity error!
336:
337:     CPU -> Store(CPU -> AssemblyChannel -> Select(
338:         (MoveMode ?
339:             TAssemblyChannel::AsmChannelWMB :
340:             TAssemblyChannel::AsmChannelWMA),
341:             TAssemblyChannel::AsmChannelZonesA,
342:             false,
343:             TAssemblyChannel::AsmChannelSignNone,
344:             TAssemblyChannel::AsmChannelNumA) );
345:
346:     // If we have bad data, but not asterisk insert, STOP
347:     // One good way: Run FORTRAN w/o Asterisk Insert! ;)
348:
349:     if(!CPU -> AChannel -> Select().CheckParity()) {
350:         CPU -> AChannelCheck -> SetStop("Data Check, No Asterisk Insert!");
351:         return;
352:     }
353:
354:     ResetR2();                                // Reset Channel Data
355:     if( CPU -> StorageWrapCheck(+1)) {        // If storage wrap -- done
356:         IntEndofTransfer = true;
357:         EndofRecord = true;
358:     }
359:     else {
360:         addr -> Set(CPU -> STARMOD(1));      // Bump address register
361:     }
362:
363: } // End, !LastInputCycle
364:
365: // In the real world, the device keeps reading data, and will
366: // then set CycleRequired. (In fact, the console matrix will
367: // call ChannelStrobe to actually do this when a key is pressed).
368: // But most of our input devices in the emulator are co-routines,
369: // so we have to call them back to give them a chance to strobe
370: // the channel again. Also, in the real world, a device would
371: // keep reading input data after the channel set Internal End of
372: // Transfer -- we have to give the input co-routine a way to
373: // finish reading it's record, even if we are no longer storing
374: // it because we hit a GMWM or wrapped storage.
375:
376: CurrentDevice -> DoInput();
377:
378: }
379:
380: bool T1410Channel::ChannelStrobe(BCD ch) {
381:
382:     // If R1 has data already, move it to R2.
383:     // (In the emulator, that should probably never happen!)
384:
385:     if(GetR1Status()) {
386:         MoveR1R2();
387:     }
388:
389:     // Load R1, and copy to R2 if there is room in R2.
390:
391:     SetR1(ch);
392:     if(!GetR2Status()) {
393:         MoveR1R2();
394:     }
395:
396:     // If the channel has not already terminated the transfer,

```

Q Grubbs

col } ①

```
397:     // ask to send the data to memory.
398:
399:     if(!IntEndofTransfer) {
400:         CycleRequired = true;
401:     }
402:
403:     InputRequest = true;                                // Force co-routine call
404:     return true;
405: }
406:
407: // Channel Unit Control just passes it off to the device.
408:
409: void T1410Channel::DoUnitControl(BCD opmod) {
410:     GetCurrentDevice() -> DoUnitControl(opmod);
411:     if(ExtEndofTransfer) {
412:         IntEndofTransfer = true;
413:     }
414:     return;
415: }
416:
417: □
418:
419: // Class T1410IODevice implementation. This is an *abstract* base class,
420: // intended to be used to derive actual I/O devices
421:
422: T1410IODevice::T1410IODevice(int devicenumber, T1410Channel *Ch) {
423:     Channel = Ch;
424:     Ch -> AddIODevice(this,devicenumber);
425: }
426:
427: // Method to handle Unit Control for those devices which don't have unit
428: // control (most of them)
429:
430: void T1410IODevice::DoUnitControl(BCD opmod) {
431:     Channel -> SetStatus(Channel -> GetStatus() | IOCHNOTREADY);
432:     DEBUG("Unit control not implemented for device",0);
433:     return;
434: }
435:
436:
437:
438: // And, finally, the 1410 IO Instruction Routines. We implement them
439: // here to keep the I/O stuff together!
440:
441: // Move and Load mode (M and L) Instructions.
442:
443: // Note: The channel selected by the CPU is known to be available, as the
444: // interlock test was passed during instruction readout at I3.
445: // IOChannelSelect indicates the selected channel.
446: // Channel -> ChUnitType has Device Type (e.g. 'T' for console)
447: // Channel -> ChUnitNumber has Unit Number
448:
449: void T1410CPU::InstructionIO() {
450:
451:     BCD opmod;
452:     T1410Channel *Ch = Channel[IOChannelSelect];
453:
454:     opmod = (Op_Mod_Reg -> Get().To6Bit());
455:     assert(!(Ch -> ChInterlock -> State()));
456:
457:     // Reset the channel, then set Channel Interlock
458:
459:     Ch -> Reset();
460:     Ch -> ChInterlock -> Set();
461:
462:     // Set Move mode or Load mode, appropriately
```

```

463:
464:     if((Op_Reg -> Get()).To6Bit() == OP_IO_MOVE) {
465:         Ch -> MoveMode = true;
466:     }
467:     else {
468:         Ch -> LoadMode = true;
469:     }
470:
471: // Start things out, depending on op modifier.
472:
473: switch(opmod.ToInt()) {
474:
475:     case OP_MOD_SYMBOL_R:
476:     case OP_MOD_SYMBOL_DOLLAR:
477:         Ch -> ChRead -> Set();
478:         break;
479:
480:     case OP_MOD_SYMBOL_W:
481:     case OP_MOD_SYMBOL_X:
482:         Ch -> ChWrite -> Set();
483:         break;
484:
485:     default:
486:         InstructionCheck ->
487:             SetStop("Instruction Check: Invalid I/O d-character");
488:         return;
489: } // End switch on op modifier
490:
491: // See if there is a device for this device number. If not,
492: // return not ready.
493:
494: if(Ch -> GetCurrentDevice() == NULL) {
495:     Ch -> SetStatus(IOCHNOTREADY);
496:     IRingControl = true;
497:     return;
498: }
499:
500: // Start up the I/O, do initial status check (Status Sample A)
501: // Just return if the status is not 0.
502:
503: Ch -> SetStatus(Ch -> GetCurrentDevice() -> Select());
504: if(Ch -> GetStatus() != 0) {
505:     IRingControl = true;
506:     return;
507: }
508:
509: // If OK so far, set Overlap/NotOverlap
510:
511: if(CPU -> IOOverlapSelect) {
512:     Ch -> ChOverlap -> Set();
513:     // TODO: Ch -> RequestOutput() or RequestInput (Write/Read)
514:     IRingControl = true;
515:     return;
516: }
517: else {
518:     Ch -> ChNotOverlap -> Set();
519: }
520:
521: // If we get here, we are not overlapped.
522:
523: if((opmod.ToInt() & OP_MOD_SYMBOL_W) == OP_MOD_SYMBOL_W) {
524:     CPU -> Display();
525:     while(!Ch -> ExtEndofTransfer) {
526:         Ch -> DoOutput(B_AR);
527:     }
528:     Ch -> ChNotOverlap -> Reset();

```

①

~~Bad logic~~

~~Overlap logic~~

```

529:     Ch -> SetStatus(Ch -> GetCurrentDevice() -> StatusSample());
530:     IRingControl = true;
531:     return;
532: }
533: else if((opmod.ToInt() & OP_MOD_SYMBOL_R) == OP_MOD_SYMBOL_R) {
534:     // Input processing continues so long as not External End from device
535:     CPU -> Display();      ↙ Ch -> GetCurrentDevice → DoInput();
536:     while(!Ch -> ExtEndofTransfer) {
537:         // If no input, just wait here (not overlapped -- stuck here)
538:         while(!(Ch -> CycleRequired || Ch -> InputRequest ||
539:                 Ch -> ExtEndofTransfer)) {
540:             Application -> ProcessMessages();
541:             // sleep(10);
542:             continue;
543:         }
544:         Ch -> DoInput(B_AR);  (Calls Device DoInput())
545:     } // End, not External End of Transfer
546:     assert(Ch -> ExtEndofTransfer);
547: }
548: // If, at the end, things do not match up ==> Wrong Length Record
549: // (Note that we do *not* test InputRequest here!)
550: // nt Cll Ch Not Overlap
551: // nt Cll Ch Not Overlap
552: // nt Cll Ch Not Overlap
553: // nt Cll Ch Not Overlap
554: // nt Cll Ch Not Overlap
555: // nt Cll Ch Not Overlap
556: // nt Cll Ch Not Overlap
557: // nt Cll Ch Not Overlap
558: Ch -> SetStatus(Ch -> GetCurrentDevice() -> StatusSample());
559: if(Ch -> CycleRequired || Ch -> GetR2Status() || !Ch -> EndofRecord) {
560:     Ch -> SetStatus(Ch -> GetStatus() | IOCHWLRECORD);
561: }
562: // And all done - continue with Instructions
563: Ch -> ChNotOverlap -> Reset();
564: IRingControl = true;
565: return;
566: }
567: }
568: }
569: }
570: }
571: }
572: // Unit control (U) Instruction. Basically, it just passes it on to
573: // the appropriate unit. The rest of the code is similar to the normal
574: // I/O instructions.
575: }
576: void T1410CPU::InstructionUnitControl() {
577:     BCD opmod;
578:     T1410Channel *Ch = Channel[IOChannelSelect];
579:     // We assume that the channel is valid, and not interlocked.
580:     // assert(Ch != NULL);
581:     assert(!Ch -> ChInterlock -> State());
582:     // The unit control function is defined by the d-character.
583:     opmod = Op_Mod_Reg -> Get().To6Bit();
584:     // Start of an I/O operation. Reset the Channel, and set Interlocked.
585:     // Check that the device requested exists, and select it. If anything
586:     // goes wrong, return.
587:     Ch -> Reset();
588:     Ch -> SetStatus(Ch -> GetCurrentDevice() -> StatusSample());
589:     IRingControl = true;
590:     // nt Cll Ch Not Overlap
591:     // nt Cll Ch Not Overlap
592:     // nt Cll Ch Not Overlap
593:     // nt Cll Ch Not Overlap
594:     Ch -> Reset();

```

Console - no EET  
Tager gnu EET

```
595:     Ch -> ChInterlock -> Set();
596:     if(Ch -> GetCurrentDevice() == NULL) {
597:         Ch -> SetStatus(IOCHNOTREADY);
598:         IRingControl = true;
599:         return;
600:     }
601:     Ch -> SetStatus(Ch -> GetCurrentDevice() -> Select());
602:     if(Ch -> GetStatus() != 0) {
603:         IRingControl = true;
604:         return;
605:     }
606:
607: // Set Overlap/Not Overlap in progress, appropriately. If Overlapped,
608: // fire things up, then return to the instruction stream.
609:
610:     if(CPU -> IOOverlapSelect) {
611:         Ch -> ChOverlap -> Set();
612:         // To Do: Start the I/O operation...
613:         IRingControl = true;
614:         return;
615:     }
616:     else {
617:         Ch -> ChNotOverlap -> Set();
618:     }
619:
620: // If we get here, we are not overlapped...
621:
622:     Ch -> DoUnitControl(opmod);
623:
624: // All we can do now is wait...
625:
626:     while(!Ch -> ExtEndofTransfer) {
627:         Application -> ProcessMessages();
628:         // sleep(10);
629:     }
630:
631: // Finish up the operation...
632:
633:     Ch -> ChNotOverlap -> Reset();
634:     Ch -> SetStatus(Ch -> GetCurrentDevice() -> StatusSample());
635:     IRingControl = true;
636:     return;
637: }
```



Write 100 - 101 (100)  
 IRG 110  
 BSP 110 - 101  
 Read 101 - 110

Back up after write(0)

(\*)  
 Write 189 → 98 (99)  
 BSP → 98 (IRG & 89)  
 write 0x80 99 (100)  
 Read 99 → 98 (IRG & 89) (100)  
 BSP 89 → 90 (IRG & 89)  
 100 → 90 (IRG & 89)

Write 93 - 102 (103\*)

BSP  
 (0x80 & 103, (104))  
 103 → 94 (IRG & 92)

Read (93) 94 - 103 (104)

OOPS → switch  
 Read → write  
 Needs IRG b.f

```
1: //-----
2: #include <vcl.h>
3: #pragma hdrstop
4:
5: #include <dir.h>
6: #include <stdio.h>
7: #include <assert.h>
8: #include "UBCD.H"
9: #include "UI1410CPUH.H"
10: #include "UITAPEUNIT.h"
11:
12: //-----
13: #pragma package(smart_init)
14:
15: #include "UI1410DEBUG.H"
16:
17: #define TAPEDEBUG 1
18:
19: // Tape Unit Implementation.
20:
21: // Constructor
22:
23: TTapeUnit::TTapeUnit(int u) {
24:
25:     fd = NULL;
26:     BusyEntry = new TBusyDevice(); // Create a busy list entry
27:     Init(u); // Let common init take over
28: }
29:
30: // Initialization (not sure if anyone else will ever use this)
31:
32: void TTapeUnit::Init(int u) {
33:
34:     if(fd != NULL) {
35:         delete fd;
36:     }
37:     irg-read = false;
38:     unit = u;
39:     fd = NULL;
40:     irg-loaded = fileprotect = tapeindicate = ready = selected = bot = false;
41:     modified = false;
42:     highdensity = true;
43:     filename[0] = '\0';
44:     record_number = 0;
45:     BusyEntry -> SetBusy(0); // Set not busy.
46:     return;
47: }
48:
49: // Methods that interface to user interface buttons
50:
51: bool TTapeUnit::Reset() {
52:     ready = false;
53:     return(true);
54: }
55:
56: // Load the tape (file) (if not already loaded) and rewind.
57:
58: bool TTapeUnit::LoadRewind() {
59:
60:     if(ready) { // Inop if drive is ready
61:         return(false);
62:     }
63:     writing
64:     if(modified && fd) {
65:         assert(fd != NULL);
66:         Write(0);
```

```
67:         modified = false;
68:     }
69:
70:     if(fd != NULL) {                                // If loaded, just rewind
71:
72:         try {
73:             fd -> Seek(0, soFromBeginning);
74:             fd = modified = false;           writing = true;
75:             record_number = 0;
76:             return(bot = loaded = true);    ready = false;
77:         }
78:
79:         catch(char *dummy) {
80:             DEBUG("LoadRewind: Seek on failed on tape unit %d",unit);
81:             delete fd;
82:             fd = NULL;
83:             return(false);
84:         }
85:
86:     }
87:     else if(strlen(filename) == 0) {
88:         return(false);
89:     }
90:
91:     // Open the file. First try RW. If that fails, try RO and set fileprot.
92:
93:     try {
94:         fd = new TFileStream(filename, fmOpenReadWrite);
95:         fileprotect = false;
96:     }
97:
98:     catch(EFOpenError &e) {
99:
100:        try {
101:            fd = new TFileStream(filename, fmOpenRead);
102:            fileprotect = true;           // Read Only was OK
103:        }
104:
105:        catch(EFOpenError &e) {
106:            DEBUG("LoadRewind: open failed on tape unit %d",unit);
107:            DEBUG(e.Message.c_str(),0);
108:            ready = loaded = fileprotect = bot = modified = false;
109:            return(false);
110:        }
111:    }
112:
113:    irg = modified = false;      writing = true;
114:    record_number = 0;
115:    return(bot = loaded = true);
116: }
117:
118: // Unload the tape (file)
119:
120: bool TTapeUnit::Unload() {
121:
122:     if(ready || !loaded) {                         // If ready, ignore.
123:         return(false);
124:     }
125:
126:     if(fd != NULL) {
127:         File delete fd;
128:         fd = NULL;
129:     }
130:     irg = loaded = fileprotect = bot = modified = false;  irg-read } = false
131:     tapeindicate = false;
132:     record_number = 0;
133:
```

```
133:     return(true);
134: }
135:
136: // Mount a tape on the drive (associate a file)
137:
138: bool TTapeUnit::Mount(char *fname) {
139:
140:     if(ready || loaded) {                                // If ready or already
141:         return(false);                                // loaded, ignore it.
142:     }
143:
144:     if(strlen(fname) == 0 || strlen(fname)+1 > sizeof(filename)) {
145:         return(false);
146:     }
147:     assert(fd == NULL);
148:     strcpy(filename,fname);
149:     irg = fileprotect = tapeindicate = bot = modified = false; writing } -false;
150:     return(true);
151: }
152:
153: // Start Button
154:
155: bool TTapeUnit::Start() {
156:
157:     if(ready || !loaded) {                                // If ready or not loaded
158:         return(false);                                // can't help you!
159:     }
160:
161:     assert(fd != NULL);
162:
163:     return(ready = true);
164: }
165:
166: bool TTapeUnit::ChangeDensity() {
167:     if(ready) {
168:         return(false);
169:     }
170:     highdensity = !highdensity;
171:     return(true);
172: }
173:
174: // Methods that interface with the Tape Adapter Unit (TAU)
175:
176: bool TTapeUnit::Select(bool b) {
177:     selected = b;
178:     return(true);
179: }
180:
181: // Rewind to beginning of tape (file)
182:
183: bool TTapeUnit::Rewind() {
184:
185:     if(!selected || !loaded || !ready) {                // Must be ready to go...
186:         DEBUG("TTapeUnit::Rewind: Unit %d not selected or not ready",unit);
187:         return(false);
188:     }
189:
190: #ifdef TAPEDEBUG
191:     DEBUG("Rewind unit %d",unit);
192: #endif
193:
194:     assert(fd != NULL); writing
195:
196:     if(modified && irg) {                                // Mark end of record
197:         Write(0);
198:         modified = false;
test
```

```
199:     }
200:
201:     try {
202:         fd -> Seek(0, soFromBeginning);
203:     }
204:
205:     catch(char *dummy) {
206:         DEBUG("Rewind: Seek failed, unit %d",unit);
207:         delete fd;
208:         fd = NULL;
209:         loaded = ready = bot = false; writing } = false;
210:         return(false);
211:     }
212:
213:     if(modified = false, writing = true, irg.read = false);
214:     BusyEntry -> SetBusy(record_number); // Act like we are busy
215:     record_number = 0;
216:     return(bot = true);
217: }
218:
219: // Rewind and unload the tape (close the file)
220:
221: bool TTapeUnit::RewindUnload() {
222:
223:     if(!Rewind()) { // If rewind fails,
224:         delete fd;
225:         fd = NULL; Rewind();
226:         return(false);
227:     }
228:
229:     if(fd != NULL) {
230:         delete fd;
231:         fd = NULL;
232:
233:         Rewind();
234:         if(!loaded = fileprotect = ready = bot = false, writing } = false;
235:             tapeindicate = false;
236:             record_number = 0;
237:             return(true);
238:     }
239:
240: // Skip and blank tape, does nothing for now (until we have measured tape)
241:
242: bool TTapeUnit::Skip() {
243:
244: #ifdef TAPEDEBUG
245:     DEBUG("Write IRG (Skip) unit %d",unit);
246: #endif
247:
248:     if(!selected || !loaded || !ready) {
249:         return(false);
250:     }
251:     return(true);
252: }
253:
254: // Space forward. (d-character is "A" - not in my Principles of Operation!)
255: // Basically pretty easy: all we have to do is call read until we get a
256: // negative return code, which will happen at EOF or IRG or an error.
257: // We can let the Tape Adapter Unit figure out the status.
258:
259: int TTapeUnit::Space() {
260:
261:     int rc;
262:
263: #ifdef TAPEDEBUG
264:     DEBUG("Space unit %d",unit);
```

```
265: #endif
266:
267:     while((rc = Read()) >= 0) {
268:         // Do nothing.
269:     }
270:     BusyEntry -> SetBusy(1);           // Must go busy for a while
271:     return(rc);
272: }
273:
274: // Backspace. This one is a pain. To do it, we take two steps back, one
275: // forward. Repeatedly. Slow. Oh well....
276:
277: bool TTapeUnit::Backspace() {
278:
279:     if(!selected || !ready || !loaded) {
280:         DEBUG("TTapeUnit::Backspace: Unit %d not selected or not ready",unit);
281:         return(false);
282:     }
283:
284:     assert(fd != NULL);
285:
286: #ifdef TAPEDBEG
287:     DEBUG("Backspace start: %d",fd -> Position);
288: #endif
289:
290:     if(bot) {                         // If at BOT, a NOP
291:         record_number = 0;
292:         return(true);
293:     }
294:
295:     BusyEntry -> SetBusy(1);           // If not a BOT, go busy
296:
297:     // If we just ended a record, write out its IRG, then back up before it.
298:
299:     if(modified && irg) {           writing
300:         Write(0);
301:         modified = irg = false;
302:         try {                      writing = irg-read = false;
303:             fd -> Seek(-1,soFromCurrent);
304:         }
305:
306:
307:         catch(char *dummy) {
308:             DEBUG("Backspace: Seek over EOR failed on unit %d",unit);
309:             delete fd;
310:             fd = NULL;
311:             irg = loaded = fileprotect = ready = false; writing } = false;
312:             return(false);
313:         }
314:     }
315:
316:
317:     // Now, go into the two steps back, one step forward routine.
318:
319:     while(true) {                   // Start the dance...
320:
321:         // Seek back 2 characters
322:
323:         try {
324:             fd -> Seek(-2,soFromCurrent);
325:         }
326:
327:         catch(char *dummy) {
328:             DEBUG("Backspace: Seek failed on unit %d",unit);
329:             delete fd;
330:             fd = NULL;
```

```

331:     if(rg->loaded && fileprotect && ready) {
332:         irg = loaded = fileprotect = ready = false;
333:     }
334:
335:     // If beginning of file, done! (special case)
336:
337:     if(fd->Position == 0) {
338:         irg = false;
339:         bot = true;
340:         writing record_number = 0;
341: #ifdef TAPEDEBUG
342:         DEBUG("Backspace end at BOT",0);
343: #endif
344:         return(true);
345:     }
346:
347:     // Read forward 1 character. Quit on error or EOF
348:
349:     if(fd->Read(&tape_buffer,1) != 1) {
350:         DEBUG("Backspace: Read failed: unexpected eof on unit %d",unit);
351:         return(false);
352:     }
353:
354:     // If we find an IRG bit on, we are done! Have to leave the IRG
355:     // bit on the character, though.
356:
357:     if(tape_buffer & TAPE_IRG) {
358:         irg = true;
359:         --record_number;
360: #ifdef TAPEDEBUG
361:         DEBUG("Backspace end: %d",fd->Position);
362: #endif
363:         return(true);
364:     }
365: }
366:
367:
368: // Method to write a character. Note that by this time any cute stuff
369: // (like parity, changing wordmarks in to word separators, changing
370: // word separators into two word separators, etc. should have already been
371: // handled in the TAU
372:
373: bool TTapeUnit::Write(char c) {
374:
375:     // If we have an irg left over from a previous read, back up over it.
376:
377:     if(irg && modified) { . . . irg-read
378:         try {
379:             DEBUG("Write seeking back over EOR from: %d",fd->Position);
380:             fd->Seek(-1,soFromCurrent);
381:             DEBUG("Write seeking back over EOR to: %d",fd->Position);
382:             writing = modified = true; // Set modified to write IRG
383:         }
384:
385:         catch(char *dummy) {
386:             DEBUG("Write: Seek over EOR failed on unit %d",unit);
387:             delete fd;
388:             fd = NULL;
389:             irg = loaded = fileprotect = ready = false;
390:             return(false);
391:         }
392:     }
393:
394:     // If we have an IRG left to do from the previous write, set the IRG bit.
395:
396:     if(irg && modified) {
397:         writing
398:     }
399:
400:     if(irg && modified) {
401:         writing
402:     }
403:
404:     if(irg && modified) {
405:         writing
406:     }
407:
408:     if(irg && modified) {
409:         writing
410:     }
411:
412:     if(irg && modified) {
413:         writing
414:     }
415:
416:     if(irg && modified) {
417:         writing
418:     }
419:
420:     if(irg && modified) {
421:         writing
422:     }
423:
424:     if(irg && modified) {
425:         writing
426:     }
427:
428:     if(irg && modified) {
429:         writing
430:     }
431:
432:     if(irg && modified) {
433:         writing
434:     }
435:
436:     if(irg && modified) {
437:         writing
438:     }
439:
440:     if(irg && modified) {
441:         writing
442:     }
443:
444:     if(irg && modified) {
445:         writing
446:     }
447:
448:     if(irg && modified) {
449:         writing
450:     }
451:
452:     if(irg && modified) {
453:         writing
454:     }
455:
456:     if(irg && modified) {
457:         writing
458:     }
459:
460:     if(irg && modified) {
461:         writing
462:     }
463:
464:     if(irg && modified) {
465:         writing
466:     }
467:
468:     if(irg && modified) {
469:         writing
470:     }
471:
472:     if(irg && modified) {
473:         writing
474:     }
475:
476:     if(irg && modified) {
477:         writing
478:     }
479:
480:     if(irg && modified) {
481:         writing
482:     }
483:
484:     if(irg && modified) {
485:         writing
486:     }
487:
488:     if(irg && modified) {
489:         writing
490:     }
491:
492:     if(irg && modified) {
493:         writing
494:     }
495:
496:     if(irg && modified) {
497:         writing
498:     }
499:
500:     if(irg && modified) {
501:         writing
502:     }
503:
504:     if(irg && modified) {
505:         writing
506:     }
507:
508:     if(irg && modified) {
509:         writing
510:     }
511:
512:     if(irg && modified) {
513:         writing
514:     }
515:
516:     if(irg && modified) {
517:         writing
518:     }
519:
520:     if(irg && modified) {
521:         writing
522:     }
523:
524:     if(irg && modified) {
525:         writing
526:     }
527:
528:     if(irg && modified) {
529:         writing
530:     }
531:
532:     if(irg && modified) {
533:         writing
534:     }
535:
536:     if(irg && modified) {
537:         writing
538:     }
539:
540:     if(irg && modified) {
541:         writing
542:     }
543:
544:     if(irg && modified) {
545:         writing
546:     }
547:
548:     if(irg && modified) {
549:         writing
550:     }
551:
552:     if(irg && modified) {
553:         writing
554:     }
555:
556:     if(irg && modified) {
557:         writing
558:     }
559:
560:     if(irg && modified) {
561:         writing
562:     }
563:
564:     if(irg && modified) {
565:         writing
566:     }
567:
568:     if(irg && modified) {
569:         writing
570:     }
571:
572:     if(irg && modified) {
573:         writing
574:     }
575:
576:     if(irg && modified) {
577:         writing
578:     }
579:
580:     if(irg && modified) {
581:         writing
582:     }
583:
584:     if(irg && modified) {
585:         writing
586:     }
587:
588:     if(irg && modified) {
589:         writing
590:     }
591:
592:     if(irg && modified) {
593:         writing
594:     }
595:
596:     if(irg && modified) {
597:         writing
598:     }
599:
600:     if(irg && modified) {
601:         writing
602:     }
603:
604:     if(irg && modified) {
605:         writing
606:     }
607:
608:     if(irg && modified) {
609:         writing
610:     }
611:
612:     if(irg && modified) {
613:         writing
614:     }
615:
616:     if(irg && modified) {
617:         writing
618:     }
619:
620:     if(irg && modified) {
621:         writing
622:     }
623:
624:     if(irg && modified) {
625:         writing
626:     }
627:
628:     if(irg && modified) {
629:         writing
630:     }
631:
632:     if(irg && modified) {
633:         writing
634:     }
635:
636:     if(irg && modified) {
637:         writing
638:     }
639:
640:     if(irg && modified) {
641:         writing
642:     }
643:
644:     if(irg && modified) {
645:         writing
646:     }
647:
648:     if(irg && modified) {
649:         writing
650:     }
651:
652:     if(irg && modified) {
653:         writing
654:     }
655:
656:     if(irg && modified) {
657:         writing
658:     }
659:
660:     if(irg && modified) {
661:         writing
662:     }
663:
664:     if(irg && modified) {
665:         writing
666:     }
667:
668:     if(irg && modified) {
669:         writing
670:     }
671:
672:     if(irg && modified) {
673:         writing
674:     }
675:
676:     if(irg && modified) {
677:         writing
678:     }
679:
680:     if(irg && modified) {
681:         writing
682:     }
683:
684:     if(irg && modified) {
685:         writing
686:     }
687:
688:     if(irg && modified) {
689:         writing
690:     }
691:
692:     if(irg && modified) {
693:         writing
694:     }
695:
696:     if(irg && modified) {
697:         writing
698:     }
699:
700:     if(irg && modified) {
701:         writing
702:     }
703:
704:     if(irg && modified) {
705:         writing
706:     }
707:
708:     if(irg && modified) {
709:         writing
710:     }
711:
712:     if(irg && modified) {
713:         writing
714:     }
715:
716:     if(irg && modified) {
717:         writing
718:     }
719:
720:     if(irg && modified) {
721:         writing
722:     }
723:
724:     if(irg && modified) {
725:         writing
726:     }
727:
728:     if(irg && modified) {
729:         writing
730:     }
731:
732:     if(irg && modified) {
733:         writing
734:     }
735:
736:     if(irg && modified) {
737:         writing
738:     }
739:
740:     if(irg && modified) {
741:         writing
742:     }
743:
744:     if(irg && modified) {
745:         writing
746:     }
747:
748:     if(irg && modified) {
749:         writing
750:     }
751:
752:     if(irg && modified) {
753:         writing
754:     }
755:
756:     if(irg && modified) {
757:         writing
758:     }
759:
760:     if(irg && modified) {
761:         writing
762:     }
763:
764:     if(irg && modified) {
765:         writing
766:     }
767:
768:     if(irg && modified) {
769:         writing
770:     }
771:
772:     if(irg && modified) {
773:         writing
774:     }
775:
776:     if(irg && modified) {
777:         writing
778:     }
779:
780:     if(irg && modified) {
781:         writing
782:     }
783:
784:     if(irg && modified) {
785:         writing
786:     }
787:
788:     if(irg && modified) {
789:         writing
790:     }
791:
792:     if(irg && modified) {
793:         writing
794:     }
795:
796:     if(irg && modified) {
797:         writing
798:     }
799:
800:     if(irg && modified) {
801:         writing
802:     }
803:
804:     if(irg && modified) {
805:         writing
806:     }
807:
808:     if(irg && modified) {
809:         writing
810:     }
811:
812:     if(irg && modified) {
813:         writing
814:     }
815:
816:     if(irg && modified) {
817:         writing
818:     }
819:
820:     if(irg && modified) {
821:         writing
822:     }
823:
824:     if(irg && modified) {
825:         writing
826:     }
827:
828:     if(irg && modified) {
829:         writing
830:     }
831:
832:     if(irg && modified) {
833:         writing
834:     }
835:
836:     if(irg && modified) {
837:         writing
838:     }
839:
840:     if(irg && modified) {
841:         writing
842:     }
843:
844:     if(irg && modified) {
845:         writing
846:     }
847:
848:     if(irg && modified) {
849:         writing
850:     }
851:
852:     if(irg && modified) {
853:         writing
854:     }
855:
856:     if(irg && modified) {
857:         writing
858:     }
859:
860:     if(irg && modified) {
861:         writing
862:     }
863:
864:     if(irg && modified) {
865:         writing
866:     }
867:
868:     if(irg && modified) {
869:         writing
870:     }
871:
872:     if(irg && modified) {
873:         writing
874:     }
875:
876:     if(irg && modified) {
877:         writing
878:     }
879:
880:     if(irg && modified) {
881:         writing
882:     }
883:
884:     if(irg && modified) {
885:         writing
886:     }
887:
888:     if(irg && modified) {
889:         writing
890:     }
891:
892:     if(irg && modified) {
893:         writing
894:     }
895:
896:     if(irg && modified) {
897:         writing
898:     }
899:
900:     if(irg && modified) {
901:         writing
902:     }
903:
904:     if(irg && modified) {
905:         writing
906:     }
907:
908:     if(irg && modified) {
909:         writing
910:     }
911:
912:     if(irg && modified) {
913:         writing
914:     }
915:
916:     if(irg && modified) {
917:         writing
918:     }
919:
920:     if(irg && modified) {
921:         writing
922:     }
923:
924:     if(irg && modified) {
925:         writing
926:     }
927:
928:     if(irg && modified) {
929:         writing
930:     }
931:
932:     if(irg && modified) {
933:         writing
934:     }
935:
936:     if(irg && modified) {
937:         writing
938:     }
939:
940:     if(irg && modified) {
941:         writing
942:     }
943:
944:     if(irg && modified) {
945:         writing
946:     }
947:
948:     if(irg && modified) {
949:         writing
950:     }
951:
952:     if(irg && modified) {
953:         writing
954:     }
955:
956:     if(irg && modified) {
957:         writing
958:     }
959:
960:     if(irg && modified) {
961:         writing
962:     }
963:
964:     if(irg && modified) {
965:         writing
966:     }
967:
968:     if(irg && modified) {
969:         writing
970:     }
971:
972:     if(irg && modified) {
973:         writing
974:     }
975:
976:     if(irg && modified) {
977:         writing
978:     }
979:
980:     if(irg && modified) {
981:         writing
982:     }
983:
984:     if(irg && modified) {
985:         writing
986:     }
987:
988:     if(irg && modified) {
989:         writing
990:     }
991:
992:     if(irg && modified) {
993:         writing
994:     }
995:
996:     if(irg && modified) {
997:         writing
998:     }
999:
1000:    if(irg && modified) {
1001:        writing
1002:    }
1003:
1004:    if(irg && modified) {
1005:        writing
1006:    }
1007:
1008:    if(irg && modified) {
1009:        writing
1010:    }
1011:
1012:    if(irg && modified) {
1013:        writing
1014:    }
1015:
1016:    if(irg && modified) {
1017:        writing
1018:    }
1019:
1020:    if(irg && modified) {
1021:        writing
1022:    }
1023:
1024:    if(irg && modified) {
1025:        writing
1026:    }
1027:
1028:    if(irg && modified) {
1029:        writing
1030:    }
1031:
1032:    if(irg && modified) {
1033:        writing
1034:    }
1035:
1036:    if(irg && modified) {
1037:        writing
1038:    }
1039:
1040:    if(irg && modified) {
1041:        writing
1042:    }
1043:
1044:    if(irg && modified) {
1045:        writing
1046:    }
1047:
1048:    if(irg && modified) {
1049:        writing
1050:    }
1051:
1052:    if(irg && modified) {
1053:        writing
1054:    }
1055:
1056:    if(irg && modified) {
1057:        writing
1058:    }
1059:
1060:    if(irg && modified) {
1061:        writing
1062:    }
1063:
1064:    if(irg && modified) {
1065:        writing
1066:    }
1067:
1068:    if(irg && modified) {
1069:        writing
1070:    }
1071:
1072:    if(irg && modified) {
1073:        writing
1074:    }
1075:
1076:    if(irg && modified) {
1077:        writing
1078:    }
1079:
1080:    if(irg && modified) {
1081:        writing
1082:    }
1083:
1084:    if(irg && modified) {
1085:        writing
1086:    }
1087:
1088:    if(irg && modified) {
1089:        writing
1090:    }
1091:
1092:    if(irg && modified) {
1093:        writing
1094:    }
1095:
1096:    if(irg && modified) {
1097:        writing
1098:    }
1099:
1100:   if(irg && modified) {
1101:       writing
1102:   }
1103:
1104:   if(irg && modified) {
1105:       writing
1106:   }
1107:
1108:   if(irg && modified) {
1109:       writing
1110:   }
1111:
1112:   if(irg && modified) {
1113:       writing
1114:   }
1115:
1116:   if(irg && modified) {
1117:       writing
1118:   }
1119:
1120:   if(irg && modified) {
1121:       writing
1122:   }
1123:
1124:   if(irg && modified) {
1125:       writing
1126:   }
1127:
1128:   if(irg && modified) {
1129:       writing
1130:   }
1131:
1132:   if(irg && modified) {
1133:       writing
1134:   }
1135:
1136:   if(irg && modified) {
1137:       writing
1138:   }
1139:
1140:   if(irg && modified) {
1141:       writing
1142:   }
1143:
1144:   if(irg && modified) {
1145:       writing
1146:   }
1147:
1148:   if(irg && modified) {
1149:       writing
1150:   }
1151:
1152:   if(irg && modified) {
1153:       writing
1154:   }
1155:
1156:   if(irg && modified) {
1157:       writing
1158:   }
1159:
1160:   if(irg && modified) {
1161:       writing
1162:   }
1163:
1164:   if(irg && modified) {
1165:       writing
1166:   }
1167:
1168:   if(irg && modified) {
1169:       writing
1170:   }
1171:
1172:   if(irg && modified) {
1173:       writing
1174:   }
1175:
1176:   if(irg && modified) {
1177:       writing
1178:   }
1179:
1180:   if(irg && modified) {
1181:       writing
1182:   }
1183:
1184:   if(irg && modified) {
1185:       writing
1186:   }
1187:
1188:   if(irg && modified) {
1189:       writing
1190:   }
1191:
1192:   if(irg && modified) {
1193:       writing
1194:   }
1195:
1196:   if(irg && modified) {
1197:       writing
1198:   }
1199:
1200:   if(irg && modified) {
1201:       writing
1202:   }
1203:
1204:   if(irg && modified) {
1205:       writing
1206:   }
1207:
1208:   if(irg && modified) {
1209:       writing
1210:   }
1211:
1212:   if(irg && modified) {
1213:       writing
1214:   }
1215:
1216:   if(irg && modified) {
1217:       writing
1218:   }
1219:
1220:   if(irg && modified) {
1221:       writing
1222:   }
1223:
1224:   if(irg && modified) {
1225:       writing
1226:   }
1227:
1228:   if(irg && modified) {
1229:       writing
1230:   }
1231:
1232:   if(irg && modified) {
1233:       writing
1234:   }
1235:
1236:   if(irg && modified) {
1237:       writing
1238:   }
1239:
1240:   if(irg && modified) {
1241:       writing
1242:   }
1243:
1244:   if(irg && modified) {
1245:       writing
1246:   }
1247:
1248:   if(irg && modified) {
1249:       writing
1250:   }
1251:
1252:   if(irg && modified) {
1253:       writing
1254:   }
1255:
1256:   if(irg && modified) {
1257:       writing
1258:   }
1259:
1260:   if(irg && modified) {
1261:       writing
1262:   }
1263:
1264:   if(irg && modified) {
1265:       writing
1266:   }
1267:
1268:   if(irg && modified) {
1269:       writing
1270:   }
1271:
1272:   if(irg && modified) {
1273:       writing
1274:   }
1275:
1276:   if(irg && modified) {
1277:       writing
1278:   }
1279:
1280:   if(irg && modified) {
1281:       writing
1282:   }
1283:
1284:   if(irg && modified) {
1285:       writing
1286:   }
1287:
1288:   if(irg && modified) {
1289:       writing
1290:   }
1291:
1292:   if(irg && modified) {
1293:       writing
1294:   }
1295:
1296:   if(irg && modified) {
1297:       writing
1298:   }
1299:
1300:   if(irg && modified) {
1301:       writing
1302:   }
1303:
1304:   if(irg && modified) {
1305:       writing
1306:   }
1307:
1308:   if(irg && modified) {
1309:       writing
1310:   }
1311:
1312:   if(irg && modified) {
1313:       writing
1314:   }
1315:
1316:   if(irg && modified) {
1317:       writing
1318:   }
1319:
1320:   if(irg && modified) {
1321:       writing
1322:   }
1323:
1324:   if(irg && modified) {
1325:       writing
1326:   }
1327:
1328:   if(irg && modified) {
1329:       writing
1330:   }
1331:
1332:   if(irg && modified) {
1333:       writing
1334:   }
1335:
1336:   if(irg && modified) {
1337:       writing
1338:   }
1339:
1340:   if(irg && modified) {
1341:       writing
1342:   }
1343:
1344:   if(irg && modified) {
1345:       writing
1346:   }
1347:
1348:   if(irg && modified) {
1349:       writing
1350:   }
1351:
1352:   if(irg && modified) {
1353:       writing
1354:   }
1355:
1356:   if(irg && modified) {
1357:       writing
1358:   }
1359:
1360:   if(irg && modified) {
1361:       writing
1362:   }
1363:
1364:   if(irg && modified) {
1365:       writing
1366:   }
1367:
1368:   if(irg && modified) {
1369:       writing
1370:   }
1371:
1372:   if(irg && modified) {
1373:       writing
1374:   }
1375:
1376:   if(irg && modified) {
1377:       writing
1378:   }
1379:
1380:   if(irg && modified) {
1381:       writing
1382:   }
1383:
1384:   if(irg && modified) {
1385:       writing
1386:   }
1387:
1388:   if(irg && modified) {
1389:       writing
1390:   }
1391:
1392:   if(irg && modified) {
1393:       writing
1394:   }
1395:
1396:   if(irg && modified) {
1397:       writing
1398:   }
1399:
1400:   if(irg && modified) {
1401:       writing
1402:   }
1403:
1404:   if(irg && modified) {
1405:       writing
1406:   }
1407:
1408:   if(irg && modified) {
1409:       writing
1410:   }
1411:
1412:   if(irg && modified) {
1413:       writing
1414:   }
1415:
1416:   if(irg && modified) {
1417:       writing
1418:   }
1419:
1420:   if(irg && modified) {
1421:       writing
1422:   }
1423:
1424:   if(irg && modified) {
1425:       writing
1426:   }
1427:
1428:   if(irg && modified) {
1429:       writing
1430:   }
1431:
1432:   if(irg && modified) {
1433:       writing
1434:   }
1435:
1436:   if(irg && modified) {
1437:       writing
1438:   }
1439:
1440:   if(irg && modified) {
1441:       writing
1442:   }
1443:
1444:   if(irg && modified) {
1445:       writing
1446:   }
1447:
1448:   if(irg && modified) {
1449:       writing
1450:   }
1451:
1452:   if(irg && modified) {
1453:       writing
1454:   }
1455:
1456:   if(irg && modified) {
1457:       writing
1458:   }
1459:
1460:   if(irg && modified) {
1461:       writing
1462:   }
1463:
1464:   if(irg && modified) {
1465:       writing
1466:   }
1467:
1468:   if(irg && modified) {
1469:       writing
1470:   }
1471:
1472:   if(irg && modified) {
1473:       writing
1474:   }
1475:
1476:   if(irg && modified) {
1477:       writing
1478:   }
1479:
1480:   if(irg && modified) {
1481:       writing
1482:   }
1483:
1484:   if(irg && modified) {
1485:       writing
1486:   }
1487:
1488:   if(irg && modified) {
1489:       writing
1490:   }
1491:
1492:   if(irg && modified) {
1493:       writing
1494:   }
1495:
1496:   if(irg && modified) {
1497:       writing
1498:   }
1499:
1500:   if(irg && modified) {
1501:       writing
1502:   }
1503:
1504:   if(irg && modified) {
1505:       writing
1506:   }
1507:
1508:   if(irg && modified) {
1509:       writing
1510:   }
1511:
1512:   if(irg && modified) {
1513:       writing
1514:   }
1515:
1516:   if(irg && modified) {
1517:       writing
1518:   }
1519:
1520:   if(irg && modified) {
1521:       writing
1522:   }
1523:
1524:   if(irg && modified) {
1525:       writing
1526:   }
1527:
1528:   if(irg && modified) {
1529:       writing
1530:   }
1531:
1532:   if(irg && modified) {
1533:       writing
1534:   }
1535:
1536:   if(irg && modified) {
1537:       writing
1538:   }
1539:
1540:   if(irg && modified) {
1541:       writing
1542:   }
1543:
1544:   if(irg && modified) {
1545:       writing
1546:   }
1547:
1548:   if(irg && modified) {
1549:       writing
1550:   }
1551:
1552:   if(irg && modified) {
1553:       writing
1554:   }
1555:
1556:   if(irg && modified) {
1557:       writing
1558:   }
1559:
1560:   if(irg && modified) {
1561:       writing
1562:   }
1563:
1564:   if(irg && modified) {
1565:       writing
1566:   }
1567:
1568:   if(irg && modified) {
1569:       writing
1570:   }
1571:
1572:   if(irg && modified) {
1573:       writing
1574:   }
1575:
1576:   if(irg && modified) {
1577:       writing
1578:   }
1579:
1580:   if(irg && modified) {
1581:       writing
1582:   }
1583:
1584:   if(irg && modified) {
1585:       writing
1586:   }
1587:
1588:   if(irg && modified) {
1589:       writing
1590:   }
1591:
159
```

Mark

```
397:         c |= TAPE_IRG;
398:     }
399:
400: (1) { 400: if(!loaded || !ready || tapeindicate || !selected || fd == NULL) {
401:     DEBUG("TapeUnit::Write: Unit %d not ready or selected",unit);
402:     return(false);
403: }
404:
405: } 405: if(fileprotect) {
406:     DEBUG("TapeUnit::Write: Attempt to write when file protected, unit %d",
407:           unit);
408:     return(false);
409: }
410:
411: #ifdef TAPEDeBUG
412:     DEBUG("Write start: %d",fd -> Position);
413: #endif
414:
415:     if(fd -> Write(&c,1) != 1) {
416:         DEBUG("TapeUnit::Write: File I/O error writing on unit %d",unit);
417:         delete fd;
418:         fd = NULL;
419:         tapeindicate = true;
420:         ready = loaded = fileprotect = bot = raise;
421:         return(false);
422:     }
423:     writing = ingread = false;
424:     fd = false;
425:
426: #ifdef TAPEDeBUG
427:     DEBUG("Write end: %d",fd -> Position);
428: #endif
429:
430:     return(true);
431: }
432:
433: // Mark end of record. Called at the end of a transfer by the TAU.
434: // All this does is set the IRG flag for the start of the next record.
435:
436: void TTapeUnit::WriteIRG() {
437:     irg = modified = true;
438:     bot = false;
439:     ++record_number;
440:
441: #ifdef TAPEDeBUG
442:     DEBUG("Write IRG unit %d",unit);
443: #endif
444:
445:     return;
446: }
447:
448: // Write tape mark. Just calls write to do the dirty work...
449: // Note that tape marks are *always* even parity. Since this writes
450: // an IRG and flushes, it also clears the modified flag.
451:
452: bool TTapeUnit::WriteTM() {
453:     bool status;
454:
455:     ++record_number;
456:
457: #ifdef TAPEDeBUG
458:     DEBUG("Write TM %d",unit);
459: #endif
460:
461:     if(!Write(TAPE_TM | TAPE_IRG)) {
462:         return(false);
463:     }
464:
```

```

463:     }
464:     irg = modified = bot = false; {writing = true}
465:     status = Write(TAPE_TM); {irg-read = false}
466:
467: #ifdef TAPEDEBUG
468:     DEBUG("Write TM end: %d", fd->Position);
469: #endif
470:
471:     return(status);
472: }
473:
474: // Read a character. Has to handle one interesting case. The tapes
475: // have an extra Tape Mark for each tape mark, so we have to skip that
476: // extra one. (The 2nd Tape Mark does NOT have IRG set).
477:
478: int TTapeUnit::Read() {
479:
480:     int rc;
481:
482:     if(!loaded || !ready || !selected || tapeindicate || fd == NULL) {
483:         DEBUG("TapeUnit::Read: Unit not ready or selected: %d", unit);
484:         return(TAPEUNITNOTREADY);
485:     }
486:
487:     // Read a character, unless the last read resulted in an IRG, in
488:     // which case it is already in the buffer.
489:     // Interesting statuses are negative, just bubble them on up.
490:
491:     if(!irg) { irg-read
492:         if((rc = ReadNextChar()) < 0) {
493:             return(rc);
494:         }
495:         tape_buffer = (char) rc;
496:     }
497:
498:     // If at an irg and the character read is a TM, skip bogus next
499:     // char (or chars) until next IRG, and return EOF with Indicate.
500:     // A Tape Mark is only a Tape Mark as the first character. Also,
501:     // when we first see it at the end of a record, it just denotes the
502:     // IRG. (Which means we need to save it in tape_buffer, too)
503:     irg-read
504:     if((bot || irg) &&
505:         (tape_buffer & 0x3f) == TAPE_TM &&
506:         (tape_buffer & TAPE_IRG) != 0) {
507: #ifdef TAPEDEBUG
508:         DEBUG("Tape Mark found, offset %d", fd->Position);
509: #endif
510:         while((rc = ReadNextChar()) >= 0 && (rc & TAPE_IRG) == 0) {
511:             // Skip chars until next IRG.
512:         }
513:         ++record_number;
514:         if(rc >= 0) {
515:             tape_buffer = (char) rc; irg-read
516:             bot = true;
517:         }
518:         tapeindicate = true;
519:         bot = false;
520:         return(TAPEUNITEOF);
521:     }
522:
523:     // If we are at bot or an IRG, strip the IRG bit from the char.
524:     // We will then just return that char.
525:     irg-read
526:     if(bot || irg) {
527:         tape_buffer &= (~TAPE_IRG);
528:         bot = irg = false;
529:         irg-read

```

```
529:     }
530:
531:     // If we hit the end of the record, then set irg now, and return such
532:
533:     if(tape_buffer & TAPE_IRG) {
534: #ifdef TAPEDEBUG
535:         DEBUG("TTapeUnit::Read: Found IRG character at %d", fd -> Position);
536: #endif
537:     irg = true;    irg-read
538:     bot = false;
539:     ++record_number;
540:     return(TAPEUNITIRG);
541: }
542:
543: // Aw shucks, just return the blinkin' character already!
544:
545: return(tape_buffer); irg=false;
546: } irg-read
547:
548: // Utility method to read a character from the file, and handle a few odds
549: // and ends. Keeps us from having to do it all more than once...
550:
551: int TTapeUnit::ReadNextChar() {
552:
553:     unsigned char c;
554:
555: #ifdef TAPEDEBUG
556:     DEBUG("Tape Read start: %d", fd -> Position);
557: #endif
558:
559:     if(fd->Read(&c,1) != 1) {
560:         DEBUG("TapeUnit::ReadNextChar: Error or EOF in Read, unit %d",unit);
561:         tapeindicate = true;
562:         return(TAPEUNITEOF);
563:     }
564:
565:     return(c);
566: }
567:
568:
```

```
1: //-----
2: #include <vcl.h>
3: #pragma hdrstop
4:
5: #include <dir.h>
6: #include <stdio.h>
7: #include <assert.h>
8: #include <errno.h>
9: #include "UBCD.H"
10: #include "UI1410CPUUT.H"
11: #include "UITAPEUNIT.h"
12:
13: //-----
14: #pragma package(smart_init)
15:
16: #include "UI1410DEBUG.H"
17:
18: // Tape Unit Implementation.
19:
20: // Constructor
21:
22: TTapeUnit::TTapeUnit(int u) {
23:
24:     fd = NULL;
25:     BusyEntry = new TBusyDevice();           // Create a busy list entry
26:     Init(u);                             // Let common init take over
27: }
28:
29: // Initialization (not sure if anyone else will ever use this)
30:
31: void TTapeUnit::Init(int u) {
32:
33:     if(fd != NULL) {
34:         fclose(fd);
35:     }
36:
37:     unit = u;
38:     fd = NULL;
39:     loaded = fileprotect = tapeindicate = ready = selected = bot = false;
40:     modified = false;
41:     highdensity = true;
42:     filename[0] = '\0';
43:     record number = 0;
44:     BusyEntry -> SetBusy(0);               // Set not busy.
45:     return;
46: }
47:
48: // Methods that interface to user interface buttons
49:
50: bool TTapeUnit::Reset() {
51:     ready = false;
52:     return(true);
53: }
54:
55: // Load the tape (file) (if not already loaded) and rewind.
56:
57: bool TTapeUnit::LoadRewind() {
58:
59:     if(ready) {                           // Inop if drive is ready
60:         return(false);
61:     }
62:
63:     if(modified && irg) {
64:         assert(fd != NULL);
65:         Write(0);
66:         fflush(fd);                      // Flush output.
```

```
67:         modified = false;
68:     }
69:
70:     if(fd != NULL) {                                // If loaded, just rewind
71:         if(fseek(fd,0L,SEEK_SET) < 0) {
72:             DEBUG("LoadRewind: fseek on failed on tape unit %d",unit);
73:             DEBUG("LoadRewind: errno is %d",errno);
74:             fclose(fd);
75:             fd = NULL;
76:             return(false);
77:         }
78:     }
79:     else if(strlen(filename) == 0) {
80:         return(false);
81:     }
82:
83:     // Open the file. First try RW. If that fails, try RO and set fileprot.
84:
85:     if((fd = fopen(filename,"rb+")) == NULL) {        // Open RW. OK?
86:         if((fd = fopen(filename,"rb")) == NULL) {      // No. How about RO?
87:             DEBUG("LoadRewind: fopen failed on tape unit %d",unit);
88:             DEBUG("LoadRewind: errno is %d",errno);
89:             ready = loaded = fileprotect = bot = irg = false;
90:             return(false);
91:         }
92:         else {                                         // RO OK
93:             fileprotect = true;
94:         }
95:     }
96:     else {                                         // RW OK
97:         fileprotect = false;
98:     }
99:
100:    irg = modified = false;
101:    record_number = 0;
102:    return(bot = loaded = true);
103: }
104:
105: // Unload the tape (file)
106:
107: bool TTapeUnit::Unload() {
108:
109:     if(ready || !loaded) {                          // If ready, ignore.
110:         return(false);
111:     }
112:
113:     if(fd != NULL) {
114:         fclose(fd);
115:         fd = NULL;
116:     }
117:     irg = loaded = fileprotect = bot = modified = false;
118:     tapeindicate = false;
119:     record_number = 0;
120:     return(true);
121: }
122:
123: // Mount a tape on the drive (associate a file)
124:
125: bool TTapeUnit::Mount(char *fname) {
126:
127:     if(ready || loaded) {                          // If ready or already
128:         return(false);                            // loaded, ignore it.
129:     }
130:
131:     if(strlen(fname) == 0 || strlen(fname)+1 > sizeof(filename)) {
132:         return(false);
```

```
133:     }
134:     assert(fd == NULL);
135:     strcpy(filename, fname);
136:     fileprotect = tapeindicate = bot = modified = false;
137:     return(true);
138: }
139:
140: // Start Button
141:
142: bool TTapeUnit::Start() {
143:     if(readv || !loaded) {                                // Tf ready or not loaded
144:         if(!loaded)
145:             readv = true;
146:         else
147:             readv = false;
148:     }
149: }
```

145: return(false); // can't help you!

146: }

147:

148: assert(fd != NULL);

149:

150: return(ready = true);

151: }

152:

153: bool TTapeUnit::ChangeDensity() {

154: if(ready) {

155: return(false);

156: }

157: highdensity = !highdensity;

158: return(true);

159: }

160:

161: // Methods that interface with the Tape Adapter Unit (TAU)

162:

163: bool TTapeUnit::Select(bool b) {

164: selected = b;

165: return(true);

166: }

167:

168: // Rewind to beginning of tape (file)

169:

170: bool TTapeUnit::Rewind() {

171:

172: if(!selected || !loaded || !ready) { // Must be ready to go...

173: DEBUG("TTapeUnit::Rewind: Unit %d not selected or not ready",unit);

174: return(false);

175: }

176:

177: assert(fd != NULL);

178:

179: if(modified && ~~irg~~) { // Mark end of record

180: Write(0);

181: fflush(fd);

182: modified = false;

183: }

184:

185: if(fseek(fd,0L,SEEK\_SET) < 0) {

186: DEBUG("Rewind: fseek failed, unit %d",unit);

187: DEBUG("Rewind: errno is %d",errno);

188: fd = NULL;

189: ~~irg~~ loaded = ready = bot = false;

190: return(false);

191: }

192:

193: ~~irg~~ modified = false;

194: BusyEntry -> SetBusy(record\_number); // Act like we are busy

195: record\_number = 0;

196: return(bot = true);

197: }

198:

```
199: // Rewind and unload the tape (close the file)
200:
201: bool TTapeUnit::RewindUnload() {
202:
203:     if(!Rewind()) {                                // If rewind fails,
204:         fclose(fd);                            // close the file anyway
205:         fd = NULL;
206:         return(false);
207:     }
208:
209:     if(fd != NULL) {
210:         fclose(fd);
211:         fd = NULL;
212:     }
213:
214:     irg = loaded = fileprotect = ready = bot = false;
215:     tapeindicate = false;
216:     record_number = 0;
217:     return(true);
218: }
219:
220: // Skip and blank tape, does nothing for now (until we have measured tape)
221:
222: bool TTapeUnit::Skip() {
223:
224:     if(!selected || !loaded || !ready) {
225:         return(false);
226:     }
227:     modified = true;
228:     return(true);
229: }
230:
231: // Space forward. (d-character is "A" - not in my Principles of Operation!)
232: // Basically pretty easy: all we have to do is call read until we get a
233: // negative return code, which will happen at EOF or IRG or an error.
234: // We can let the Tape Adapter Unit figure out the status.
235:
236: int TTapeUnit::Space() {
237:
238:     int rc;
239:
240:     while((rc = Read()) >= 0) {
241:         // Do nothing.
242:     }
243:     BusyEntry -> SetBusy(1);                      // Must go busy for a while
244:     return(rc);
245: }
246:
247: // Backspace. This one is a pain. To do it, we take two steps back, one
248: // forward. Repeatedly. Slow. Oh well....
249:
250: bool TTapeUnit::Backspace() {
251:
252:     if(!selected || !ready || !loaded) {
253:         DEBUG("TTapeUnit::Backspace: Unit %d not selected or not ready",unit);
254:         return(false);
255:     }
256:
257:     assert(fd != NULL);
258:
259:     if(bot) {                                     // If at BOT, a NOP
260:         record_number = 0;
261:         return(true);
262:     }
263:
264:     BusyEntry -> SetBusy(1);
```



{ if(modified && irg) } } // If not a BOT, go busy  
if(modified && write(0)) {  
 modified = false; } // If at BOT, a NOP

```

265:     fflush(fd);
266:
267:     while(true) {                                     // Start the dance...
268:
269:         // Seek back 2 characters
270:
271:         if(fseek(fd,-2L,SEEK_CUR) != 0) {
272:             DEBUG("Backspace: lseek failed on unit %d",unit);
273:             DEBUG("Backspace: errno is %d",errno);
274:             fclose(fd);
275:             fd = NULL;
276:             irg = loaded = fileprotect = ready = false;
277:             return(false);
278:         }
279:
280:         // If beginning of file, done! (special case)
281:
282:         if(ftell(fd) == 0L) {
283:             irg = false;
284:             bot = true;
285:             record_number = 0;
286:             return(true);
287:         }
288:
289:         // Read forward 1 character.  Quit on error or EOF
290:
291:         if(fread(&tape_buffer,1,1,fd) != 1) {
292:             if(!feof(fd)) {
293:                 DEBUG("Backspace: fread failed on unit %d",unit);
294:                 DEBUG("Backspace: errno is %d",errno);
295:                 fclose(fd);
296:                 fd = NULL;
297:                 irg = loaded = fileprotect = ready = false;
298:                 return(false);
299:             }
300:             DEBUG("Backspace: fread failed: unexpected eof on unit %d",unit);
301:             return(false);
302:         }
303:
304:         // If we find an IRG bit on, we are done!
305:
306:         if(tape_buffer & TAPE_IRG) {
307:             tape_buffer ^= TAPE_IRG;           no (has to have - could be TM)
308:             irg = true;
309:             --record_number;
310:             return(true);
311:         }
312:     }
313: }
314:
315: // Method to write a character.  Note that by this time any cute stuff
316: // (like parity, changing wordmarks in to word separators, changing
317: // word separators into two word separators, etc. should have already been
318: // handled in the TAU
319:
320: bool TTapeUnit::Write(char c) {
321:
322:     if(irg) {
323:         c |= TAPE_IRG;
324:     }
325:
326:     if(!loaded || !ready || tapeindicate || !selected || fd == NULL) {
327:         DEBUG("TapeUnit::Write: Unit %d not ready or selected",unit);
328:         return(false);
329:     }
330:
```

```
331:     if(fileprotect) {
332:         DEBUG("TapeUnit::Write: Attempt to write when file protected, unit %d",
333:               unit);
334:         return(false);
335:     }
336:
337:     if(fwrite(&c,1,1,fd) != 1) {
338:         DEBUG("TapeUnit::Write: File I/O error writing on unit %d",unit);
339:         DEBUG("TapeUnit::Write: errno is %d",errno);
340:         fclose(fd);
341:         fd = NULL;
342:         tapeindicate = true;
343:         irg ready = loaded = fileprotect = bot = false;
344:         return(false);
345:     }
346:
347:     irg = false;
348:     return(true);
349: }
350:
351: // Mark end of record. Called at the end of a transfer by the TAU.
352: // All this does is set the IRG flag for the start of the next record.
353:
354: void TTapeUnit::WriteIRG() {
355:     irg modified = true;
356:     bot = false;
357:     fflush(fd);
358:     ++record_number;
359:     return;
360: }
361:
362: // Write tape mark. Just calls write to do the dirty work...
363: // Note that tape marks are *always* even parity
364:
365: bool TTapeUnit::WriteTM() {
366:     bool status;
367:
368:     ++record_number;
369:     modified = true;
370:     if(!Write(TAPE_TM | TAPE_IRG)) {
371:         fflush(fd);
372:         return(false);
373:     }
374:     status = Write(TAPE_TM);
375:     fflush(fd);
376:     return(status);
377: }
378:
379: // Read a character. Has to handle one interesting case. The tapes
380: // have an extra Tape Mark for each tape mark, so we have to skip that
381: // extra one. (The 2nd Tape Mark does NOT have IRG set).
382:
383: int TTapeUnit::Read() {
384:
385:     int rc;
386:
387:     if(!loaded || !ready || !selected || tapeindicate || fd == NULL) {
388:         DEBUG("TapeUnit::Read: Unit not ready or selected: %d",unit);
389:         return(TAPEUNITNOTREADY);
390:     }
391:
392:     // Read a character, unless the last read resulted in an IRG, in
393:     // which case it is already in the buffer.
394:     // Interesting statuses are negative, just bubble them on up.
395:
396:     if(!irg) {
```

```
397:         if((rc = ReadNextChar()) < 0) {
398:             return(rc);
399:         }
400:         tape_buffer = (char) rc;
401:     }
402:
403: // If at an irg and the character read is a TM, skip bogus next
404: // char, and return EOF indication.
405:
406: if((tape_buffer & 0x3f) == TAPE_TM && (tape_buffer & TAPE_IRG) != 0) {
407:     if((rc = ReadNextChar()) < 0) {
408:         return(rc);
409:     }
410:     ++record_number;
411:     tapeindicate = true;
412:     bot = false;
413:     return(TAPEUNITEOF);
414: }
415:
416: // If we are not at bot or an IRG, strip the IRG bit from the char.
417: // We will then just return that char.
418:
419: if(bot || irg) {
420:     tape_buffer &= (~TAPE_IRG);
421:     bot = irg = false;
422: }
423:
424: // If we hit the end of the record, then set irg now, and return such
425:
426: if(tape_buffer & TAPE_IRG) {
427:     irg = true;
428:     bot = false;
429:     ++record_number;
430:     return(TAPEUNITIRG);
431: }
432:
433: // Aw shucks, just return the blinkin' character already!
434:
435: return(tape_buffer);
436: }
437:
438: // Utility method to read a character from the file, and handle a few odds
439: // and ends. Keeps us from having to do it all more than once...
440:
441: int TTapeUnit::ReadNextChar() {
442:
443:     unsigned char c;
444:
445:     if(fread(&c,1,1,fd) != 1) {
446:         if(!feof(fd)) {
447:             DEBUG("TapeUnit::ReadNextChar: I/O Error Reading File, unit %d",unit);
448:             DEBUG("TapeUnit::ReadNextChar: errno is %d",errno);
449:             irg = loaded = fileprotect = ready = false;
450:             tapeindicate = true;
451:             return(TAPEUNITERROR);
452:         }
453:         tapeindicate = true;
454:         return(TAPEUNITEOF);
455:     }
456:
457:     return(c);
458: }
459:
```

*need  
to read  
next char  
too until  
IRG is set irg*

Write 0 - 70 ↓ ↓  
TFile  
Rewind ↓  
Rewind ↓

Rewind ↓  
~~WriteTFile~~ (skip) (cancel)

Rewind -

Space  
(Read 0 - 70 71) ↓

Rewind ↓  
Rewind skip

Rewind  
Rewind  
~~WriteTFile~~ (cancel) (skip)  
BSP ↓ (no op) GLIP

BSP ↓ (no op)  
~~WriteTFile (cancel)~~ SKIP

Space  
(Read 0 - 70 71) ↓

BSP Start ~~0~~, END 0 (OK)

~~WriteTFile Start 0~~  
~~WriteTFile End 0~~

BSP Start 0

Backup Start 0

SKIP (nonzero)

Write TM ↓  
0 1

BSP did not work

new TFileStream (name, mode)  
y

from Open Read Write

properties Position (int)

VERIFY  
Delete to char

Seek offset, <sup>if</sup> FromBeginning  
so FromCurrent

Read (& bits) count

EFOpenError

Cannot open file

BSP start 2

Space un + 1

	at	ch	type	Bugz	r#
2066	49(A)	Ø	Ø	Ø	
		Ø	Ø	2	
2091	41(R)	Ø	Ø	Ø	
		Ø	Ø	2	Ø

(2 BOT)

2149	Select	Ø	Ø	2	Ø	BWT
				2	Ø	Ø

6175	53(E)	Ø	Ø	Ø	Ø
------	-------	---	---	---	---

6975 Select

6175 41(R)

---

2181	Select	Ø	Ø	Ø	Ø
	(R)	Ø	Ø	Ø	Ø

	Status	Ø	Ø	Ø	Ø
--	--------	---	---	---	---

6975	Select	Ø	Ø	Ø	Ø
	(E)	Ø	Ø	Ø	Ø

	Status	Ø	Ø	Ø	Ø
--	--------	---	---	---	---

6175	Select	Ø	—		
	(B)	Ø	Ø	Ø	Ø

(2 BOT) } 2207

	Status	Ø	Ø	Ø	Ø
--	--------	---	---	---	---

2219	Select	Ø	—	Ø	
	B	Ø	—	Ø	

	Status	Ø	—	Ø	
--	--------	---	---	---	--

6175	Select	Ø	—	Ø	
	(E)(S3)	Ø	—	Ø	

	Status	Ø	—	Ø	
--	--------	---	---	---	--

2250	Select	Ø	—	Ø	
	(A)(A)	Ø	—	Ø	

	Status	Ø	Ø	2	1
--	--------	---	---	---	---

WTRK Ø - 7Ø

WRK 72, 72

Rewind  
SKIP

Rewind

Rewind

SKIP

SPACE

Read Ø - 7Ø (7)

Rewind

SKIP

Rewind

Rewind

SKIP

BSP (@BOT)

BSP (@BOT)

SKIP

- SPACE

Read Ø - 7Ø (7)

BSP (end @ BOT)

SKIP

BSP (@BOT)

BSP (@BOT)

SKIP

WTM Ø 1 (2)

BSP (end @ BOT)

SPACE does not work

SKIP

Write (backwards offsets?)

4/2 21:41 Busy (?) Fullfeed Leg

Generally cruddy news service

4/3 19:30 News Server Unresponsive

20:27 DNS Failure msnews.microsoft.com  
(could be their Name Server?)

4/4 8:34+19:57 - CTRL-C

4/10 20:56 News Server unresponsive  
msnews lookup failure

~~(i.e. 9/5+more) Long~~

~~Longer run time~~

~~(i.e. 12s CPU time)~~

~~DO~~

1744	Write	&
1814	RWD	&
6975	Skip (?)	&
6975	RWD	&
2035	RWD	&
6975	SKIP (?)	&
2066	Space (?)	& (?)
2092	RWD	&
2149	RWD	&
	env 01	
	env 02	

Rewind - go busy  
~~Freeze~~ & initialize

Chancery #

T1410IO Device \* Busy Device

~~Busy List~~  
Bring Photo Banks (from int)  
Add to Busy List (Count)  
Delete from Busy List  
Release from Busy Count  
Decrease Busy Count  
Check Error

Init (clears Bus)

Rewind (busy r#)

Space Bus 1

Backspace Bus 1

under CBT

2250 Space (2)

Type indicator  $\Rightarrow$  NOT Ready  
↓  
EOL  
Type Mark

2035  
2066, 2066  
2091, 2091

(11=12)

6175 x 4

2181 STATUS Bus 1

6175 x 4 Ø Ø

2217 (50=B) Ø Ø

6175 (53=E) Ø Ø

2250 (45=A) Ø 2

6175 (50=B) Ø 2, Ø, Ø

2250 A Ø 2

6175

CF 180

6ke 300  
7ke 700

Amoco	28.25
MC	295.85
Visa	—
	324.10

TCB 4070.35

Visa 2.50

Hire Ins 243  
Car Ins 339.70  
Clothes  $\rightarrow$  200  
Appl. 700+60

Dr. Rent -60  
Pysd 68.05  
Whe 163.00

7407 9878 9697

Ex Øø& Tdtb

7374

✓ 47274

✓ 414411  
✓ 49878

BRE CEST-1  
MCWA

CØ2ØB-3

✓ 6281

✓ 69696



✓ 3.

✓ Ø7?

✓ 3.

"Instruction Check:

Edit B wr Ø Zer Supres off.

CC01

CØ2ØB-2

✓ 29.13  
✓ 29.14  
29.24

CØ21

✓ 3  
✓ 29.24

✓ Opword  $\Rightarrow$  result @ IR<sub>Ind</sub> = 8

Set

Irreg 1 2addr only

Irreg 6 1addr+mod

Irreg 11 2addr+mod

if arith chained at I1

CPU takes D cycle, during which  
BAR  $\rightarrow$  STAR + a  $\rightarrow$  DAR

Code Mem  
by doing CAR  
(Fix ec)

✓ if Multiply or Divide chained at  
I1 Cycle update to CAR

ZA A+X6

X6 = 29608

S 17056 29607 11111

X3 = 29598

X5 = 29508

ZA one address - failed?

Address Check Cycles 1

6975	Select	∅	∅	1	1	(Bus 1)
6975	Select	∅	2	1	1	(Bus 1)
6975	Select	∅	2	∅	1	Ready
	SΦ(B)	∅	∅	∅	1	
	Status	∅	∅	∅	∅	Bus 4 → 1
2282	Select	∅	∅	∅	∅	
	50	∅	∅	∅	∅	
	Status	∅	∅	∅	∅	

Now

Bus 4 from → 1024 : 010

Error 10 2373

T I Test (K)

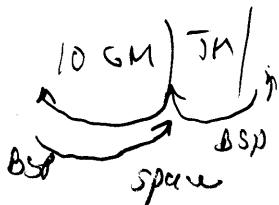
Either channel TI

BS failed

errno 19  
invalid argument

fflush fixed

Now get or 0 12, 13, 19 C 2445  
2535



TM - TI only record (if 1st ~~channel~~ channel test)

Volume in drive I is 990323\_2101  
Volume Serial Number is 7FC7-5CFc

Directory of I:\1410.021

01/19/98	01:28p	020 OK	754 UERROR.h
01/19/98	01:28p	UNTESTED	491 UERROR.cpp
01/25/98	03:14p		210 UI1410ARITH.h
01/16/99	08:32a		499 UI1410DATA.h
01/21/99	09:59p		14,697 ubcd.cpp
01/24/99	02:51p		212 UI1410BRANCH.h
01/29/99	09:50p		208 UI1410MISC.h
03/14/99	02:52p		1,436 UI1410PWR.h
03/14/99	02:52p		1,085 UI14101.h
03/14/99	02:59p		2,591 UI1415CE.h
03/14/99	03:00p		1,327 UI1415L.cpp
03/14/99	03:02p		6,873 UI1415L.h
03/14/99	03:07p		303 UI1410CPU.h
03/14/99	03:30p		708 UI1410DEBUG.cpp
03/14/99	03:33p		976 UI1410DEBUG.h
03/14/99	03:41p		1,124 UI14101.cpp
03/14/99	03:41p		10,352 UI1410CPU.cpp
03/14/99	03:42p		6,355 UI1415CE.cpp
03/14/99	03:42p		32,915 UI1410INST.cpp
03/14/99	03:42p		44,531 UI1410ARITH.cpp
03/14/99	03:42p		33,806 UI1410DATA.cpp
03/14/99	03:42p		7,296 UI1410BRANCH.cpp
03/14/99	03:42p		5,309 UI1410MISC.cpp
03/14/99	04:00p		21,060 UI1410CPU.h
03/14/99	04:13p		4,996 UI1410CHANNEL.h
03/14/99	04:19p		2,765 UI1410INST.h
03/14/99	08:19p		4,574 ubcd.h
03/15/99	07:37p		1,878 PIBM1410.cpp — add Form
03/20/99	11:17a		55,111 UI1410CPU.cpp — SetTAU x 2, Display()
03/20/99	11:22a		3,698 UI1410PWR.cpp only, if constructor ref +
03/20/99	11:49a		19,055 UI1410CHANNEL.cpp — Unit control
03/20/99	04:06p		1,536 UITAPETAU.h
03/20/99	04:21p		1,886 UI729TAPE.h
03/20/99	04:25p		2,726 UITAPEUNIT.h
03/20/99	04:29p		11,207 UITAPEUNIT.cpp
03/20/99	04:31p		5,395 UI729TAPE.cpp
03/20/99	04:58p		10,538 UITAPETAU.cpp
03/20/99	05:42p		3,659 UI1415IO.h } Busy Timer (probably)
03/20/99	05:45p		26,708 UI1415IO.cpp ) DR)
		39 File(s)	350,850 bytes
			0 bytes free

problem PIBM1410 Builds Form FI729

constructor references CPU, which doesn't  
exist yet, to init variable "Channel"

Probably, "Channel" is not really needed

MLZS 1878 0+X5 (29508)

MLCWA 1878 Ø+X6 (29608)

SBR X3 (29598) → 00039

ZA Ø+X6

1st loc'd BAR w 29608

After 2 more cyles

17056 29608 29608

ZB Ø+X6

1st loc'd A, B w/ Ø (215 words)

Then did index into B 29608 (215)

1st cycle 17056 29607 29608

2nd cycle 17056 29607 9999 ? B

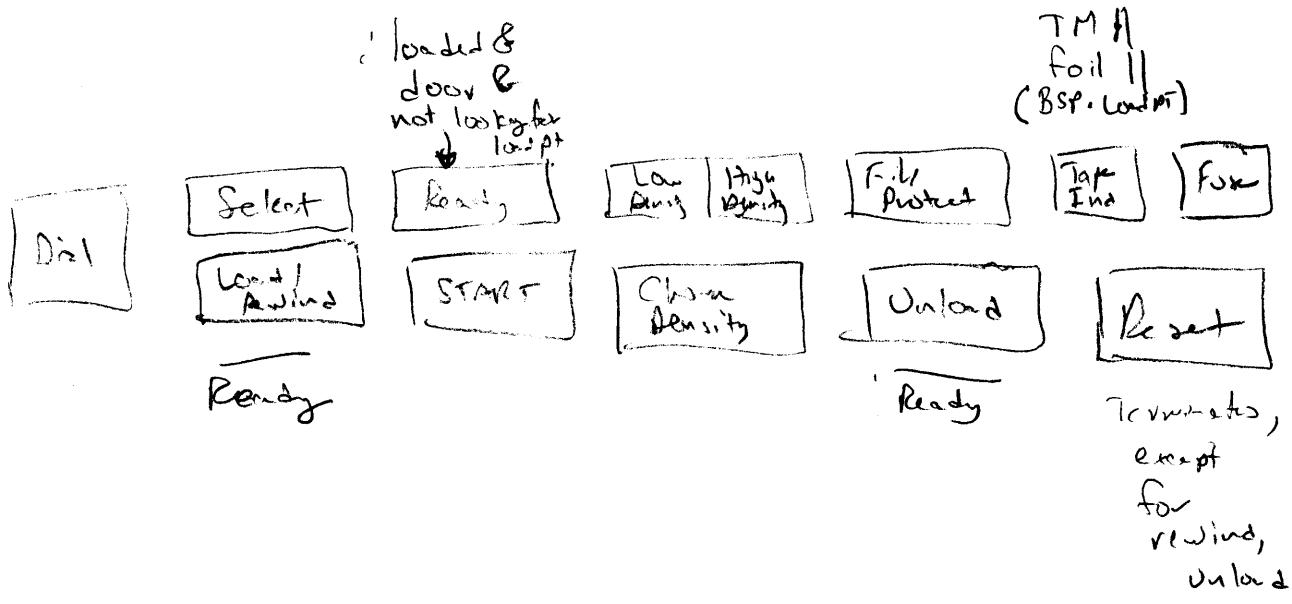
17056 29607 29608

A w

B-AR = 29608

D-AR = Ø (invalid) (!)

but units loads start from D-AR ! D  $\frac{NST}{2^k}$



7261 CONTROL

UNIT

φ

↓

?

- |                 |   |   |
|-----------------|---|---|
| ④ Select        | v | - |
| ② START / Ready | v | ✓ |
| ① Reset         | - | ✓ |
| ① LOAD / REWIND | - | ✓ |
| ① UNLOAD        | - | ✓ |
| ① FILE PROTECT  | v | - |
| ④ Tape Indirect | v | - |
| ① Low Density   | v | - |
| ② High Density  | v | - |
| ② OFF LINE      | - | - |

MOUNT

Channel.h  
cpp  
TapeUnit.h  
cpp  
Tau.h  
cpp

```
1: //-----
2: #ifndef UI729TAPEH
3: #define UI729TAPEH
4: //-----
5: #include <Classes.hpp>
6: #include <Controls.hpp>
7: #include <StdCtrls.hpp>
8: #include <Forms.hpp>
9: #include <ComCtrls.hpp>
10: //-----
11: class TFI729 : public TForm
12: {
13:     __published:    // IDE-managed Components
14:     TLabel *Unit;
15:     TUpDown *UnitDial;
16:     TButton *LoadRewind;
17:     TButton *Start;
18:     TButton *ChangeDensity;
19:     TButton *Unload;
20:     TButton *Reset;
21:     TLabel *Ready;
22:     TLabel *Select;
23:     TLabel *FileProtect;
24:     TLabel *TapeIndicate;
25:     TEdit *Filename;
26:     TButton *ChannelSelect;
27:     void __fastcall UnitDialClick(TObject *Sender, TUDBtnType Button);
28:     void __fastcall LoadRewindClick(TObject *Sender);
29:     void __fastcall StartClick(TObject *Sender);
30:     void __fastcall ChangeDensityClick(TObject *Sender);
31:     void __fastcall UnloadClick(TObject *Sender);
32:     void __fastcall ResetClick(TObject *Sender);
33:     void __fastcall ChannelSelectClick(TObject *Sender);
34: private:      // User declarations
35:
36:     int current_channel;
37:     int current_unit;
38:     T1410Channel *Channel;
39:     TTapeTAU *TAU[MAXCHANNEL];
40:     TTapeUnit *TapeUnit;
41:
42: public:       // User declarations
43:
44:     __fastcall TFI729(TComponent* Owner);
45:     void SetTAU(TTapeTAU *TAU,int channel);
46:     void Display();
47: };
48: //-----
49: extern PACKAGE TFI729 *FI729;
50: //-----
51: #endif
52:
```

Ready → Enabled

① Not mounted - should return NOT Ready + Extend of Transfer

② No data transferred either.

③ Prevents power off from running ↴ is in right place in  
read log, but neither  
Input Request or  
Cycle Required set.

④ No feedback on mount status (other than name).

⑤ Make text area R/O,

Mount invokes dialog

⑥ Use status to set buttons to show state

Start if Ready : Disable Mount, Start, Load, Unload

Load if Loaded : Disable Mount, Enable Start

Mounted : ~~Disable~~

Reset:

Unmount, Unload : Disable Start, Unload

Entered

Disable

if ready  
enable (mount, start, load, unload) enable (reset)

else if loaded

enable (mount) enable (start, ~~unload~~)  
reset ~~load~~

else (~~unloaded~~)

if file name (but unloaded)

enable load, mount

enable start, unload, reset

Mount/Unmount

else  
enable ~~all~~ all but mount

Load  
or  
start

```

1: //-----
2: #ifndef UITAPEUNITH
3: #define UITAPEUNITH
4: //-----
5:
6: // Define constants returned by Read()
7:
8: #define TAPEUNITIRG      (-1)
9: #define TAPEUNITEOF      (-2)
10: #define TAPEUNITNOTREADY (-3)          Read()
11: #define TAPEUNITERROR    (-4)
12:
13: #define TAPE_IRG        0x80
14: #define TAPE_TM         0x0f
15:
16: class TTapeUnit : public TObject {
17:
18: protected:
19:
20:     int unit;                      // Unit number.
21:
22:     char filename[MAXPATH];         // Path to tape file
23:     FILE *fd;                     // File Descriptor for I/O
24:     char tape_buffer;              // Char read from file
25:
26:     bool loaded;                  // State flags
27:     bool fileprotect;
28:     bool ready;
29:     bool selected;
30:     bool tapeindicate;
31:     bool highdensity;
32:     bool bot;
33:     bool irg;
34:
35: private:
36:     int ReadNextChar();           // Factored I/O call
37:
38:
39: public:
40:
41:     // Functions to make part of state visible to the outside
42:
43:     inline bool Selected() { return selected; }
44:     inline bool IsReady() { return ready; }
45:     inline bool IsFileProtected() { return fileprotect; }
46:     inline bool TapeIndicate() { return tapeindicate; }
47:     inline bool HighDensity() { return highdensity; }
48:     inline char *GetFileName() { return filename; }
49:
50:     // Interface functions for the User Interface buttons
51:
52:     bool LoadRewind();
53:     bool Reset();
54:     bool Unload();
55:     bool Start();
56:     bool Mount(char *filename);
57:
58:     // Interface functions for the Tape Adapter Unit (TAU) to use
59:
60:     TTapeUnit(int u);             // Constructor
61:     void Init(int u);            // Initialization
62:
63:     bool Select(bool b);         // Select or De-select unit
64:     bool Rewind();
65:     bool RewindUnload();
66:     bool Backspace();

```

*WTF*

*✓, Load() active, drop on char*

```
67:     bool Skip();           // Skip and blank tape
68:     void WriteIRG();        // Mark end of record
69:     bool WriteTM();         // Write Tape Mark (EOF)
70:
71:     int Read();            // Returns char or -value
72:     bool Write(char c);    // Write file character
73: };
74:
75: #endif
76:
```

```
1: //-----
2: #include <vcl.h>
3: #pragma hdrstop
4:
5: #include <dir.h>
6: #include <stdio.h>
7: #include <assert.h>
8: #include <errno.h>
9: #include "UITAPEUNIT.h"
10:
11: //-----
12: #pragma package(smart_init)
13:
14: #include "UI1410DEBUG.H"
15:
16: // Tape Unit Implementation.
17:
18: // Constructor
19:
20: TTapeUnit::TTapeUnit(int u) {
21:
22:     fd = NULL;
23:     Init(u);                                // Let common init take over
24: }
25:
26: // Initialization (not sure if anyone else will ever use this)
27:
28: void TTapeUnit::Init(int u) {
29:
30:     if(fd != NULL) {
31:         fclose(fd);
32:     }
33:
34:     unit = u;
35:     fd = NULL;
36:     irg = loaded = fileprotect = tapeindicate = ready = selected = bot = false;
37:     highdensity = true;
38:     filename[0] = '\0';
39:     return;
40: }
41:
42: // Methods that interface to user interface buttons
43:
44: bool TTapeUnit::Reset() {
45:     ready = false;
46:     return(true);
47: }
48:
49: // Load the tape (file) (if not already loaded) and rewind.
50:
51: bool TTapeUnit::LoadRewind() {
52:
53:     if(ready) {                                // Inop if drive is ready
54:         return(false);
55:     }
56:
57:     if(fd != NULL) {                          // If loaded, just rewind
58:         if(fseek(fd,0L,SEEK_SET) < 0) {
59:             DEBUG("LoadRewind: fseek on failed on tape unit ",unit);
60:             DEBUG("LoadRewind: errno is ",errno);
61:             fclose(fd);
62:             fd = NULL;
63:             return(false);
64:         }
65:     }
66:     else if(strlen(filename) == 0) {
```

```
67:         return(false);
68:     }
69:
70: // Open the file. First try RW. If that fails, try RO and set fileprot.
71:
72: if((fd = fopen(filename,"rb+")) == NULL) {          // Open RW. OK?
73:     if((fd = fopen(filename,"rb")) == NULL) {        // No. How about RO?
74:         DEBUG("LoadRewind: fopen failed on tape unit ",unit);
75:         DEBUG("LoadRewind: errno is ",errno);
76:         ready = loaded = fileprotect = bot = tapeindicate = irg = false;
77:         return(false);
78:     }
79:     else {                                         // RO OK
80:         fileprotect = true;
81:     }
82: }
83: else {                                         // RW OK
84:     fileprotect = false;
85: }
86:
87: irg = tapeindicate = false;
88: return(bot = loaded = true);
89: }
90:
91: // Unload the tape (file)
92:
93: bool TTapeUnit::Unload() {
94:
95:     if(ready || !loaded) {                         // If ready, ignore.
96:         return(false);
97:     }
98:
99:     if(fd != NULL) {
100:         fclose(fd);
101:         fd = NULL;
102:     }
103:     irg = loaded = fileprotect = tapeindicate = bot = false;
104:     return(true);
105: }
106:
107: // Mount a tape on the drive (associate a file)
108:
109: bool TTapeUnit::Mount(char *fname) {
110:
111:     if(ready || loaded) {                         // If ready or already
112:         return(false);                           // loaded, ignore it.
113:     }
114:
115:     if(strlen(fname) == 0 || strlen(fname)+1 > sizeof(filename)) {
116:         return(false);
117:     }
118:     assert(fd == NULL);
119:     strcpy(filename,fname);
120:     irg = fileprotect = tapeindicate = bot = false;
121:     return(true);
122: }
123:
124: // Start Button
125:
126: bool TTapeUnit::Start() {
127:
128:     if(ready || !loaded) {                         // If ready or not loaded
129:         return(false);                           // can't help you!
130:     }
131:
132:     assert(fd != NULL);
```

```
133:
134:     return(ready = true);
135: }
136:
137: // Methods that interface with the Tape Adapter Unit (TAU)
138:
139: bool TTapeUnit::Select(bool b) {
140:     selected = b;
141:     return(true);
142: }
143:
144: // Rewind to beginning of tape (file)
145:
146: bool TTapeUnit::Rewind() {
147:
148:     if(!selected || !loaded || !ready) {           // Must be ready to go...
149:         return(false);
150:     }
151:
152:     assert(fd != NULL);
153:
154:     if(fseek(fd,0L,SEEK_SET) < 0) {
155:         DEBUG("Rewind: fseek failed, unit ",unit);
156:         DEBUG("Rewind: errno is ",errno);
157:         fd = NULL;
158:         irg = loaded = ready = bot = false;
159:         return(false);
160:     }
161:
162:     irg = tapeindicate = false;
163:     return(bot = true);
164: }
165:
166: // Rewind and unload the tape (close the file)
167:
168: bool TTapeUnit::RewindUnload() {
169:
170:     if(!Rewind()) {                           // If rewind fails,
171:         fclose(fd);                         // close the file anyway
172:         fd = NULL;
173:         return(false);
174:     }
175:
176:     if(fd != NULL) {
177:         fclose(fd);
178:         fd = NULL;
179:     }
180:
181:     irg = loaded = fileprotect = ready = tapeindicate = bot = false;
182:     return(true);
183: }
184:
185: // Skip and blank tape, does nothing for now (until we have measured tape)
186:
187: bool TTapeUnit::Skip() {
188:     return(true);
189: }
190:
191: // Backspace. This one is a pain. To do it, we take two steps back, one
192: // forward. Repeatedly. Slow. Oh well....
193:
194: bool TTapeUnit::Backspace() {
195:
196:     if(!selected || !ready || !loaded) {
197:         return(false);
198:     }
```

```
199:  
200:     assert(fd != NULL);  
201:  
202:     if(bot) {  
203:         tapeindicate = true;                                // Backspace over BOT  
204:         return(true);  
205:     }  
206:  
207:     while(true) {                                         // Start the dance...  
208:  
209:         // Seek back 2 characters  
210:  
211:         if(fseek(fd,-2L,SEEK_CUR) != 0) {  
212:             DEBUG("Backspace: lseek failed on unit ",unit);  
213:             DEBUG("Backspace: errno is ",errno);  
214:             fclose(fd);  
215:             fd = NULL;  
216:             irg = loaded = fileprotect = ready = tapeindicate = false;  
217:             return(false);  
218:         }  
219:  
220:         // If beginning of file, done! (special case)  
221:  
222:         if(ftell(fd) == 0L) {  
223:             irg = false;  
224:             bot = true;  
225:             return(true);  
226:         }  
227:  
228:         // Read forward 1 character.  Quit on error or EOF  
229:  
230:         if(fread(&tape_buffer,1,1,fd) != 1) {  
231:             if(!feof(fd)) {  
232:                 DEBUG("Backspace: fread failed on unit ",unit);  
233:                 DEBUG("Backspace: errno is ",errno);  
234:                 fclose(fd);  
235:                 fd = NULL;  
236:                 irg = loaded = fileprotect = ready = tapeindicate = false;  
237:                 return(false);  
238:             }  
239:             DEBUG("Backspace: fread failed: unexpected eof on unit ",unit);  
240:             return(false);  
241:         }  
242:  
243:         // If we find an IRG bit on, we are done!  
244:  
245:         if(tape_buffer & TAPE_IRG) {  
246:             tape_buffer ^= TAPE_IRG;  
247:             irg = true;  
248:             return(true);  
249:         }  
250:     }  
251: }  
252:  
253: // Method to write a character.  Note that by this time any cute stuff  
254: // (like parity, changing wordmarks in to word separators, changing  
255: // word separators into two word separators, etc. should have already been  
256: // handled in the TAU  
257:  
258: bool TTapeUnit::Write(char c) {  
259:  
260:     if(irg) {  
261:         c |= TAPE_IRG;  
262:     }  
263:  
264:     if(!loaded || !ready || tapeindicate || fd == NULL) {
```

```

265:         DEBUG("TapeUnit::Write: Unit not ready ",unit);
266:         return(false);
267:     }
268:
269:     if(fileprotect) {
270:         DEBUG("TapeUnit::Write: Attempt to write when file protected, unit ",
271:               unit);
272:         return(false);
273:     }
274:
275:     if(fwrite(&c,1,1,fd) != 1) {
276:         DEBUG("TapeUnit::Write: File I/O error writing on unit ",unit);
277:         DEBUG("TapeUnit::Write: errno is ",errno);
278:         fclose(fd);
279:         fd = NULL;
280:         tapeindicate = true;
281:         irg = ready = loaded = fileprotect = bot = false;
282:         return(false);
283:     }
284:
285:     irg = false;
286:     return(true);
287: }
288:
289: // Mark end of record. Called at the end of a transfer by the TAU.
290: // All this does is set the IRG flag for the start of the next record.
291:
292: void TTapeUnit::WriteIRG() {
293:     irg = true;
294:     return;
295: }
296:
297: // Write tape mark. Just calls write to do the dirty work...
298: // Note that tape marks are *always* even parity
299:
300: bool TTapeUnit::WriteTM() {
301:     return(Write(TAPE_TM | TAPE_IRG));
302: }
303:
304: // Read a character. Has to handle one interesting case. The tapes
305: // have an extra Tape Mark for each tape mark, so we have to skip that
306: // extra one. (The 2nd Tape Mark does NOT have IRG set).
307:
308: int TTapeUnit::Read() {
309:
310:     int rc;
311:
312:     if(!loaded || !ready || tapeindicate || fd == NULL) {
313:         DEBUG("TapeUnit::Read: Unit not ready: ",unit);
314:         return(TAPEUNITNOTREADY);
315:     }
316:
317:     // Read a character. Interesting statuses are negative, just bubble
318:     // them on up.
319:
320:     if((rc = ReadNextChar()) < 0) {
321:         return(rc);
322:     }
323:     tape_buffer = (char) rc;
324:
325:     // If at an irg and the character read is a TM, skip bogus next char,
326:     // and return EOF indication.
327:
328:     if(tape_buffer & 0x3f tape_buffer & 0x3f) == TAPE_TM) { b (tape_buffer & TAPE_IRG != 0)
329:         if((rc = ReadNextChar()) < 0) {
330:             return(rc);

```

if(irg) {  
 irg = false;  
}  
else {  
 rc =

```
331:         }
332:         return(TAPEUNITEOF);
333:     }
334:
335: // If we are not at bot or an IRG, strip the IRG bit from the char.
336: // We will then just return that char.
337:
338: if(bot || irg) {
339:     tape_buffer &= (~TAPE_IRG);
340:     bot = irg = false;
341: }
342:
343: // If we hit the end of the record, then set irg now, and return such
344:
345: if(tape_buffer & TAPE_IRG) {
346:     irg = true;
347:     return(TAPEUNITIRG);
348: }
349:
350: // Aw shucks, just return the blinkin' character already!
351:
352: return(tape_buffer);
353: }
354:
355: // Utility method to read a character from the file, and handle a few odds
356: // and ends. Keeps us from having to do it all more than once...
357:
358: int TTapeUnit::ReadNextChar() {
359:
360:     char c;
361:
362:     if(fread(&c,1,1,fd) != 1) {
363:         if(!feof(fd)) {
364:             DEBUG("TapeUnit::ReadNextChar: I/O Error Reading File, unit ",unit);
365:             DEBUG("TapeUnit::ReadNextChar: errno is ",errno);
366:             irg = loaded = fileprotect = ready = false;
367:             tapeindicate = true;
368:             return(TAPEUNITERROR);
369:         }
370:         tapeindicate = true;
371:         return(TAPEUNITEOF);
372:     }
373:
374:     return(c);
375: }  
L8 ØX FF
```

```
1: //-----
2: #ifndef UITAPETAUH
3: #define UITAPETAUH
4: //-----
5:
6: #define UNIT_BACKSPACE 50
7: #define UNIT_SKIP 53
8: #define UNIT_REWIND 41
9: #define UNIT_REWIND_UNLOAD 20
10: #define UNIT_WTM 36
11:
12: // 1410 Tape Adapter Unit
13:
14: class TTapeTAU : public T1410IODevice {
15:
16: protected:
17:
18:     TTapeUnit *Unit[10]; // Units 0 thru 9
19:     TTapeUnit *TapeUnit; // Current selected unit
20:
21:     int tapestatus; // Channel status
22:     BCD ch_char; // Channel character
23:
24:     char tape_char; // Tape Unit character
25:     int tape_read_char; // Before checking status
26:
27:     long chars_transferred; // Storage characters
28:
29:     static char *tape_parity_table;
30:
31:
32: public:
33:
34:     // Implement the I/O device interface standard
35:
36:     TTapeTAU(int devicenum, T1410Channel *Channel);
37:     virtual int Select();
38:     virtual void DoOutput();
39:     virtual void DoInput();
40:     virtual int StatusSample();
41:     virtual void DoUnitControl(BCD opmod);
42:
43:     // State and status methods
44:
45:     TTapeUnit *GetUnit(int unit); // Select
46:
47: private:
48:
49:     // Internal methods.
50:
51:     bool DoOutputWrite(char c); // Unit 1
52:     int DoInputRead(); // Unit 2
53: };
54:
55:
56: #endif
57:
```

↓  
EOF  
NULLEAPT  
Error

d=41  
UL2  
UL1  
↓  
↓  
↓

Unit 1  
V  
U  
L

```
1: //-----
2: #include <vc1.h>
3: #pragma hdrstop
4:
5: #include <dir.h>
6: #include <stdio.h>
7: #include <assert.h>
8: #include <errno.h>
9: #include "UBCD.H"
10: #include "UI1410CPUH.H"
11: #include "UI1410CHANNEL.H"
12: #include "UITAPEUNIT.h"
13: #include "UITAPETAU.h"
14:
15: //-----
16: #pragma package(smart_init)
17:
18: // 1410 Tape Adapter Unit Implementation.  Follows I/O Device Interface.
19:
20: // Constructor.  Creates tape drives!
21: // NOTE: This is expected to be called with the EVEN PARITY designation
22: // for the device number ("U").  This constructor automatically adds the
23: // ODD PARITY designation ("B")!
24:
25: TTapeTAU::TTapeTAU(int devicenum,T1410Channel *Channel) :
26:     T1410IODevice(devicenum,Channel) {
27:
28:     static char parity_table[64] = {
29:         0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1,1,0,0,1,0,0,1,
30:         1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0
31:     };
32:
33:     int i;
34:
35:     for(i=0; i < 10; ++i) {
36:         Unit[i] = new TTapeUnit(i);
37:     }
38:
39:     TapeUnit = NULL;
40:     tapestatus = 0;
41:     chars_transferred = 0;
42:     tape_parity_table = parity_table;
43:
44:     // Create the odd parity device now...
45:
46:     Channel -> AddIODevice(this,BCD(BCD::BCDConvert('B')).To6Bit());
47: }
48:
49: // Select.  If some other unit is selected, it gets deselected.
50: // The unit comes from the CPU / Channel
51:
52: int TTapeTAU::Select() {
53:
54:     int u;
55:
56:     if(TapeUnit != NULL) {
57:         TapeUnit -> Select(false);
58:         TapeUnit = NULL;
59:     }
60:
61:     // Find the tape unit.  Give up if none there.
62:
63:     u = Channel -> GetUnitNumber();
64:     if((TapeUnit = GetUnit(u)) == NULL) {
65:         return(tapestatus = IOCHNOTREADY);
66:     }
```

```
67: // Set 0 length record so far.
68: chars_transferred = 0;
69:
70: // If writing, and tape is write protected, set not ready and condition.
71: // Not sure if that is write - the diagnostics will probably figure it out
72: // for me... 8-
73:
74: if(Channel -> ChWrite -> State() && TapeUnit -> IsFileProtected()) {
75:     return(tapestatus |= (IOCHNOTREADY | IOCHCONDITION));
76: }
77:
78: return(tapestatus = 0);
79: } → i^ reading str input
80: return(tapestatus = 0);
81: }
82:
83: // Return a pointer to a given tape drive, NULL if invalid.
84:
85: TTapeUnit *TTapeTAU::GetUnit(int u) {
86:
87:     if(u < -0 || u > 9) {
88:         return(NULL);
89:     }
90:     return(Unit[u]);
91: }
92:
93: // DoOutput: Accept an output character from the channel.
94:
95: void TTapeTAU::DoOutput() {
96:
97:     // If no (or invalid) unit selected, just return not ready.
98:
99:     if(TapeUnit == NULL) {
100:         tapestatus |= IOCHNOTREADY;
101:         Channel -> ExtEndofTransfer = true;
102:         return;
103:     }
104:
105:     // If file protected, return not ready as well. (Not sure if this is
106:     // correct, but hopefully the diagnostics will test it)
107:
108:     if(TapeUnit -> IsFileProtected()) {
109:         tapestatus |= IOCHNOTREADY;
110:         Channel -> ExtEndofTransfer = true;
111:         return;
112:     }
113:
114:     // Get the character from the channel. Strip down to 6 bits for tape.
115:
116:     ch_char = Channel -> ChR2 -> Get();
117:     tape_char = ch_char.To6Bit();
118:
119:     // If we are in load mode, and the character either has a word mark or is
120:     // itself a word separator, write out a word separator character to tape.
121:
122:     if((Channel -> LoadMode && ch_char.TestWM()) ||
123:         tape_char == (BCD_WS & 0x3f)) {
124:         if(!DoOutputWrite(BCD_WS)) {
125:             tapestatus |= IOCHCONDITION;
126:             Channel -> ExtEndofTransfer = true;
127:             return;
128:         }
129:     }
130:
131:     // If the incoming character has invalid parity, flag a datacheck. But
132:     // we will still write what we can, in valid parity, to tape.
```

```
133:     if(!ch_char.CheckParity()) {
134:         tapestatus |= IOCHDATACHECK;
135:     }
136: }
137:
138: // Finally, write out the character.
139:
140: if(!DoOutputWrite(tape_char)) {
141:     tapestatus |= IOCHCONDITION;
142:     Channel -> ExtEndofTransfer = true;
143:     return;
144: }
145:
146: ++chars_transferred;
147: }
148:
149: // Method to tell channel when we have data.
150:
151: void TTapeTAU::DoInput() {
152:
153:     bool wm = false;
154:     BCD b;
155:
156:     if(TapeUnit == NULL) {
157:         tapestatus |= IOCHNOTREADY;
158:         Channel -> ExtEndofTransfer = true;
159:         return;
160:     }
161:
162:     // If tape indicate is on, say we are not ready too.
163:
164:     if(TapeUnit -> TapeIndicate()) {
165:         tapestatus |= IOCHNOTREADY;
166:         Channel -> ExtEndofTransfer = true;
167:         return;
168:     }
169:
170:     // Get a character from the tape unit.  If an interesting status, just
171:     // return.  The utility method will have already set the right status.
172:     // (That is why the utility method exists!)
173:
174:     tape_read_char = DoInputRead();
175:     if(tape_read_char < 0) {
176:         return;
177:     }
178:
179:     ++chars_transferred;
180:
181:     // If we got a tape mark, tell the channel to store it.  The TAU always
182:     // returns tape marks in EVEN PARITY.  The channel will store it as is
183:     // (and flag a machine check) unless asterisk insert is on!
184:
185:     if((tapestatus & IOCHCONDITION) &&
186:         tape_read_char == BCD_TM) {
187:         Channel -> ChannelStrobe(BCD_TM);
188:         return;
189:     }
190:
191:     // If we are reading in load mode, and we got a word separator, read the
192:     // next character.  Then we will either set a WM on it, or, if it too is
193:     // a word separator, just store the word separator.  If we have a problem
194:     // reading the next char, just return -- the read routine will have set
195:     // status -- that is why it is there!  8-
196:
197:     if(Channel -> LoadMode && (tape_read_char & 0x3f) == (BCD_WS & 0x3f)) {
198:         wm = true;
```

```

199:     tape_read_char = DoInputRead();
200:     if(tape_read_char < 0) {
201:         return;
202:     } ← if ((tape_read_char & 0x3F) == BCD_WS) {
203:         } ←   wM = false;
204:     b = BCD(tape_read_char);
205:
206:     // Now, the character we just got is in the correct TAPE parity. If we
207:     // are reading tape in EVEN parity, we have to FLIP THE CHECK BIT!
208:
209:     if((Channel -> ChUnitType -> Get().ToInt() & 2) == 0) { // B as 2 bit - ODD
210:         // NO 2 bit -- EVEN!
211:         b.ComplementCheck();
212:     }
213:
214:     // If we set the wM flag earlier, turn it on (and flip the check bit)
215:
216:     if(wM) {
217:         b.SetWM();
218:         b.ComplementCheck();
219:     }
220:
221:
222:     // Finally, tell the channel we have something to eat.
223:
224:     Channel -> ChannelStrobe(b);
225: }
226:
227: // Method for unit control. Mostly, just passes it off to the tape unit.
228: // Also sets ExtEndofTransfer, indicating we are done. (Will probably have
229: // to work in delays someday. Oh well)
230:
231: void TTapeTAU::DoUnitControl(BCD opmod) {
232:
233:     int d;
234:
235:     d = opmod.ToInt();
236:
237:     if(TapeUnit == NULL) {
238:         tapestatus |= IOCHNOTREADY;
239:         Channel -> ExtEndofTransfer = true;
240:         return;
241:     }
242:
243:     switch(d) {
244:
245:     case UNIT_BACKSPACE:
246:         if(!TapeUnit -> Backspace()) {
247:             tapestatus |= IOCHNOTREADY;
248:         }
249:         if(TapeUnit -> TapeIndicate()) {
250:             tapestatus |= IOCHCONDITION;
251:         }
252:         break;
253:
254:     case UNIT_SKIP:
255:         if(!TapeUnit -> Skip()) {
256:             tapestatus |= IOCHNOTREADY;
257:         }
258:         break;
259:
260:     case UNIT_REWIND:
261:         if(!TapeUnit -> Rewind()) {
262:             tapestatus |= IOCHNOTREADY;
263:         }
264:         break;

```

```
265:
266:     case UNIT_REWIND_UNLOAD:
267:         if(!TapeUnit->Unload()) {
268:             tapestatus |= IOCHNOTREADY;
269:         }
270:         break;
271:
272:     case UNIT_WTM:
273:         if(!TapeUnit->WriteTM()) {
274:             tapestatus |= IOCHNOTREADY;
275:         }
276:         // Write TM in odd parity sets Data Check
277:         if(Channel->ChUnitType->Get().ToInt() & 2) {
278:             tapestatus |= IOCHDATACHECK;
279:         }
280:         break;
281:
282:     default:
283:         tapestatus |= IOCHNOTREADY;
284:         break;
285:     }
286:
287:     Channel->ExtEndofTransfer = true;
288:     return;
289: }
290:
291: // Private utility methods...
292:
293: // Utility method to write a character to the drive and collect status.
294:
295: bool TTapeTAU::DoOutputWrite(char c) {
296:
297:     int wanted_parity;
298:
299:     if(TapeUnit == NULL) {
300:         tapestatus |= IOCHNOTREADY;
301:         Channel->ExtEndofTransfer = true;
302:         return(false);
303:     }
304:
305:     c &= 0x3f;                                // Bye bye WM & Check Bit.
306:
307:     // Check the unit type. B, which has a "2" bit, means odd.
308:
309:     wanted_parity = ((Channel->ChUnitType->Get().ToInt() & 2) != 0); // 1 Odd
310:
311:     if(tape_parity_table[c] != wanted_parity) {
312:         c ^= BITC;
313:     }
314:
315:     // Finally, tell the drive to write the character.
316:
317:     return(TapeUnit->Write(c));
318: }
319:
320: // Utility method to read a character from a tape drive. Sets appropriate
321: // status if the drive returns something other than an ordinary character.
322:
323: int TTapeTAU::DoInputRead() {
324:
325:     int c;
326:     bool wanted_parity;
327:
328:     c = TapeUnit->Read();
329:
330:     if(c < 0) {
```

```
331:     Channel -> ExtEndofTransfer = true;
332:     if(c == TAPEUNITNOTREADY || c == TAPEUNITERROR) {
333:         tapestatus |= IOCHNOTREADY;
334:         return(c);
335:     }
336:     else if(c == TAPEUNITEOF) {
337:         tapestatus |= IOCHCONDITION;
338:         return(BCD_TM);
339:     }
340:     else {
341:         return(c);
342:     }
343: }
344:
345: // Check what parity we want. We want odd parity if the device in the
346: // instruction was "B" (has a 2 bit).
347:
348: wanted_parity = ((Channel -> ChUnitType -> Get().ToInt() & 2) != 0);
349:
350: if(tape_parity_table[c] != wanted_parity) {
351:     tapestatus |= IOCHDATACHECK;
352: }
353:
354: return(c);
355: }
356:
```

## TTape TAU

```
TTape Unit *Unit [10];  
TTape Unit *TapeUnit = NULL;  
int fpeStatus;  
bcd ch-char;  
char tape-char;  
long chars-transferred;  
int tape-read-char;
```

```
TTape Unit *GetUnit (int unit);  
TTape TAU (int devicenum, TI410 Channel *Channel);  
bool DoOutputWrite (char c)  
virtual int Select();  
virtual void DoOutput();  
Virtual void DoInput();  
virtual int StatusString();  
bool DoOutputWrite();  
int DoInputRead();  
virtual void DoUnitControl (BCD dchar);  
type-parity-track [ ]  
UNIT_BACKSPACE  
SKIP  
REWIND  
UNLOAD  
BCD-WS ✓  
BCD-TM
```

```
int TTapeTAU:: Select() {
    int u;
    if (TapeUnit != NULL) {
        TapeUnit->Select (false);
    }
    u = Channel->GetUnitNumber();
    TapeUnit = GetUnit(u);
    if (TapeUnit == NULL) {
        return (TapeStatus = IOCHNOTREADY);
    }
    if (Channel->ChWrite->State() &&
        TapeUnit->IsFileProtected()) {
        return (TapeStatus = IOCHNOTREADY);
    }
    return (TapeStatus = 0); } // chars transferred = 0
```

```
TTapeUnit * TTapeTAU:: GetUnit (int u) {
    if (u < 0 || u > 9) {
        return (NULL);
    }
    return Unit[u];
}
```

```

void TTapeTau :: DoOutput() {
    if (TapeUnit == NULL) {
        TapeStatus = IOCHANREADY;
        return; } Channel → ExtEndofTransfer = true;
    }

    if (TapeUnit → IsFileProtected()) {
        TapeStatus = IOCHANREADY;
        return; } Channel → ExtEndofTransfer = true;
    }

    ch-char = Channel → CHR2 → Get();
    tape-char = ch-char.To6Bit();
    if ((Channel → LoadMode && ch-char.TestWMC)) ||
        tape-char == (BCD-WS & 0x3f) ) {
        if (!DoOutputWrite(BCD-WS)) {
            TapeStatus |= IOCHCONDITION;
            return; } Channel → ExtEndofTransfer = true;
        }

    }

    if (!ch-char.CheckParity()) {
        TapeStatus |= IO DATACHECK;
        }

    if (!DoOutputWrite(tape-char)) {
        TapeStatus |= IOCHCONDITION;
        return; } Channel → ExtEndofTransfer = true;
    }

    }

    ++charsTransferred;
}

```

(1)

```

Void TTapeTau:: Do Input() {
    int odd-parity;
    bool wmr = false;
    if (Tape Unit == NULL) {
        Tape Status |= IOCHANREADY;
        Channel → ExtEndof Transfer = true;
        return;
    }

    if (Tape Unit → Tape Indicate()) {
        Tape Status |= IOCHANREADY;
        Channel → ExtEndof Transfer = true;
        return;
    }

    tape-read-char = Do Input Read();
    if (c < 0) {
        return;
    }

    if ((Tape Status & IOC1+CONDITION) != 0) {BB
        if (Tape Status == BCD-TM) {
            tape-read-char == BCD-TM;
            Channel → Channel Strobe (BCD-TM);
            return;
        }
    }

    if (Channel → Load Mode BB) {BB
        (tape-read-char & 0x3f) == BCD-W3) {
            wmr = true;
            tape-read-char = Do Input Read();
            if (c < 0) {
                return;
            }
    }
}

```

( TTapeTAU:: DoInput() )

(2)

b = tape-ready; ;

}

if (Channel → ChUnitType → Get(). ToInt() & 2) == 0 {  
    // even parity tape  
    b.ComplementCheck();

}

if (wm) {

    b.SetWM();

    b.ComplementCheck();

}

Channel → ChannelStrobe(b);

}

TTapeTAU :: TTapeTAU ( int <sup>devicenum</sup>, T1410Channel \*Channel ) :

T1410 IODevice( <sup>devicenum</sup>, Channel ) {

int i;

for (i=0; i<10; ++i) {

Unit[i] = new TTapeUnit(i);

}

LastUnit = NULL;

TapeStatus = 0;

chars\_transferred = 0;

BCD('B').To6BitC)

Channel → Add IO Device ( this, TAPE-ODD-IO\_DEVICE );

①

```

void TTapeTau:: DoUnitControl (BCD dchar) {
    int d;
    d = dchar, ToGBit();
    if (TapeUnit == NULL) {
        TapeStatus[1] = IOCHNOTREADY;
        Channel → ExtEndofTransfer = true;
        return;
    }
    switch (d) {
        case UNIT_BACKSPACE:
            if (!TapeUnit → BackSpace()) {
                TapeStatus[1] = IOCHNOTREADY;
            }
            if (TapeUnit → TapeIndicate(1)) {
                TapeStatus[1] = IOCHCONDITION;
            }
            break;
        case UNIT_SKIP:
            if (!TapeUnit → Skip()) {
                TapeStatus[1] = IOCHNOTREADY;
            }
            break;
        case UNIT_REWIND:
            if (!TapeUnit → Rewind()) {
                TapeStatus[1] = IOCHNOTREADY;
            }
            break;
    }
}

```

TTapeTAU:: DoUnitControl()

(2)

case UNIT\_REWIND\_UNLOAD:

if (!TapeUnit->UnLoad()) {

TapeStatus |= INCHNOTREADY;

}

break;

default:

TapeStatus |= INCHNOTREADY;

break;

}

Channel->ExtEndofTransfer = true;

return;

}

```
bool TTapeTau::DoOutputWrite(char c) {
```

```
    int odd-parity;
```

$$c \oplus 8 = \phi \times 3f;$$

odd-parity = (Channel → EhUnitType → Get().ToInt(), &2);

$$\therefore ! = \phi;$$

```
if (tape-parity-table[c] != odd-parity) {
```

```
    C |= TAPE-C-BST;
```

```
}
```

```
if (TapeUnit == NULL) {
```

TapeStatus |= IOCHANNOTREADY;

return false; } Channel → ExtEndOfTransfer = true;

```
}
```

```
return (TapeUnit → Write(c));
```

```
}
```

```
int TTapeTAU :: DoInputRead() {
```

```
    int c;
    bool io-odd-parity;
```

```
    c = TapeUnit->Read();
```

```
    io-parity = (Channel->ChUnitType->Get()).ToInt()
        & 2) != 0;
```

```
    if (tape-parity-table[c] != io-parity) {
        TapeStatus |= IOCHDATACHECK;
```

```
}
```

```
    Channel->ExtEndOfTransfer = true;
```

```
    if (c < 0) {
```

```
        if (c == -3) || c == -4) {
```

```
            TapeStatus |= IOCHNOTREADY;
```

```
}
```

```
        if (c == -2) {
```

```
            TapeStatus |= IOCAFCONDITION;
```

```
            return (BLD-TM);
```

```
}
```

```
        return (c);
```

```
}
```

```
int TTapeTAU:: StatusSample()  
{  
    return (Channel->GetStatus() | TapeStatus);  
}
```

```
1: //-----
2: #include <vc1.h>
3: #pragma hdrstop
4:
5: #include "UI1410CHANNEL.h"
6:
7: //-----
8: #pragma package(smart_init)
9:
10: #include <assert.h>
11: #include "UI1410CPUH.H"
12: #include "UI1410INST.H"
13: #include "UI1415CE.H"
14:
15: // Implementation of I/O Channel Class
16:
17: // Constructor.  Initializes state
18:
19: T1410Channel::T1410Channel(
20:     TLabel *LampInterlock,
21:     TLabel *LampRBCInterlock,
22:     TLabel *LampRead,
23:     TLabel *LampWrite,
24:     TLabel *LampOverlap,
25:     TLabel *LampNotOverlap,
26:     TLabel *LampNotReady,
27:     TLabel *LampBusy,
28:     TLabel *LampDataCheck,
29:     TLabel *LampCondition,
30:     TLabel *LampWLRecord,
31:     TLabel *LampNoTransfer ) {
32:
33:     int i;
34:
35:     ChStatus = 0;
36:     TapeDensity = DENSITY_200_556;
37:     R1Status = R2Status = false;
38:
39:     ChStatusDisplay[IOLAMPNOTREADY] = LampNotReady;
40:     ChStatusDisplay[IOLAMPBUSY] = LampBusy;
41:     ChStatusDisplay[IOLAMPDATACHECK] = LampDataCheck;
42:     ChStatusDisplay[IOLAMPCONDITION] = LampCondition;
43:     ChStatusDisplay[IOLAMPNOTTRANSFER] = LampNoTransfer;
44:     ChStatusDisplay[IOLAMPWLRECORD] = LampWLRecord;
45:
46:     // Generally, the channel latches are *not* reset by Program Reset
47:
48:     ChInterlock = new TDisplayLatch(LampInterlock, false);
49:     ChrBCInterlock = new TDisplayLatch(LampRBCInterlock, false);
50:     ChRead = new TDisplayLatch(LampRead, false);
51:     ChWrite = new TDisplayLatch(LampWrite, false);
52:     ChOverlap = new TDisplayLatch(LampOverlap, false);
53:     ChNotOverlap = new TDisplayLatch(LampNotOverlap, false);
54:
55:     // Generally, the channel registers are *not* reset by Program Reset
56:
57:     ChOp = new TRegister(false);
58:     ChUnitType = new TRegister(false);
59:     ChUnitNumber = new TRegister(false);
60:     ChR1 = new TRegister(false);
61:     ChR2 = new TRegister(false);
62:
63:     // Clear the device table
64:
65:     for(i=0; i < 64; ++i) {
66:         Devices[i] = NULL;
```

```
67:     }
68:
69:     CurrentDevice = NULL;
70:     MoveMode = LoadMode = false;
71:     ExtEndofTransfer = IntEndofTransfer = false;
72:     CycleRequired = false;
73:     InputRequest = false;
74:     LastInputCycle = false;
75:     EndofRecord = false;
76: }
77:
78: // Channel Register methods
79:
80: BCD T1410Channel::SetR1(BCD b) {
81:     ChR1 -> Set(b);
82:     R1Status = true;
83:     return(b);
84: };
85:
86: BCD T1410Channel::SetR2(BCD b) {
87:     ChR2 -> Set(b);
88:     R2Status = true;
89:     return(b);
90: };
91:
92: // Move data from register 1 to register 2
93: // Clearing register 1 in the process.
94:
95: BCD T1410Channel::MoveR1R2() {
96:     ChR2 -> Set(ChR1 -> Get());
97:     R1Status = false;
98:     R2Status = true;
99:     return(ChR2 -> Get());
100: }
101:
102: // Method to add a new device to the channel's device table.
103: // Typically this is called from the T1410IODevice base class constructor
104:
105: void T1410Channel::AddIODevice(T1410IODevice *iodevice, int devicenumber) {
106:     assert(Devices[devicenumber] == NULL);
107:     Devices[devicenumber] = iodevice;
108: }
109:
110: // Channel is reset during ComputerReset
111:
112: void T1410Channel::OnComputerReset()
113: {
114:     Reset();
115: }
116:
117: void T1410Channel::Reset() {
118:     ChStatus = 0;
119:     R1Status = R2Status = false;
120:     MoveMode = LoadMode = false;
121:     ExtEndofTransfer = IntEndofTransfer = false;
122:     CycleRequired = false;
123:     InputRequest = false;
124:     LastInputCycle = false;
125:     EndofRecord = false;
126:     ChInterlock -> Reset();
127:     ChRBCInterlock -> Reset();
128:     ChRead -> Reset();
129:     ChWrite -> Reset();
130:     ChOverlap -> Reset();
131:     ChNotOverlap -> Reset();
132: }
```

```
133:
134: //  Channel is not reset during Program Reset
135:
136: void T1410Channel::OnProgramReset()
137: {
138:     //  Channel not affected by Program Reset
139: }
140:
141: //  Display Routine.
142:
143: void T1410Channel::Display() {
144:
145:     int i;
146:
147:     ChStatusDisplay[IOLAMPNOTREADY] -> Enabled =
148:         ((ChStatus & IOCHNOTREADY) != 0);
149:     ChStatusDisplay[IOLAMPBUSY] -> Enabled =
150:         ((ChStatus & IOCHBUSY) != 0);
151:     ChStatusDisplay[IOLAMPDATACHECK] -> Enabled =
152:         ((ChStatus & IOCHDATACHECK) != 0);
153:     ChStatusDisplay[IOLAMPCONDITION] -> Enabled =
154:         ((ChStatus & IOCHCONDITION) != 0);
155:     ChStatusDisplay[IOLAMPWLRECORD] -> Enabled =
156:         ((ChStatus & IOCHWLRECORD) != 0);
157:     ChStatusDisplay[IOLAMPNOTTRANSFER] -> Enabled =
158:         ((ChStatus & IOCHNOTTRANSFER) != 0);
159:
160:     for(i=0; i <= 5; ++i) {
161:         ChStatusDisplay[i] -> Repaint();
162:     }
163:
164:     //  Although in most instances the following would be redundant,
165:     //  because these objects are also on the CPU display list, we include
166:     //  them here in case we want to display a channel separately.
167:
168:     ChInterlock -> Display();
169:     ChrBCInterlock -> Display();
170:     ChRead -> Display();
171:     ChWrite -> Display();
172:     ChOverlap -> Display();
173:     ChNotOverlap -> Display();
174: }
175:
176: //  Channel Lamp Test
177:
178: void T1410Channel::LampTest(bool b)
179: {
180:     int i;
181:
182:     //  Note, we don't have to do anything to the TDisplayLatch objects in
183:     //  the channel for lamp test.  They will take care of themselves on a
184:     //  lamp test.
185:
186:     if(!b) {
187:         for(i=0; i <= 5; ++i) {
188:             ChStatusDisplay[i] -> Enabled = true;
189:             ChStatusDisplay[i] -> Repaint();
190:         }
191:     }
192:     else {
193:         Display();
194:     }
195: }
196:
197: //  Channel Output to Device
198: //  Conditions on Entry: B data register (B_REG) contains character to output
```

## UI1410CHANNEL.cpp

```

199: // Channel Register 1 (E1 or F1) should be empty.
200:
201: void T1410Channel::DoOutput(TAddressRegister *addr) {
202:
203:     BCD tempbcd;
204:
205:     CPU -> CycleRing -> Set(this == CPU -> Channel[CHANNEL1] ? CYCLE_E : CYCLE_F);
206:
207:     // Read out the next character. Also, check for a storage wrap, which will
208:     // end the transfer after this character.
209:
210:     *(CPU -> STAR) = *addr;
211:     CPU -> Readout();
212:     CPU -> StorageWrapCheck(+1);
213:     tempbcd = CPU -> B_Reg -> Get();
214:
215:     // If we have a GMWM, then we are done transferring data from memory
216:     // Otherwise, Clear the WM from the input data if in Move mode, and
217:     // then (regardless of mode) transfer the data to Channel register R1
218:     // Also, if we actually have read a character out, check for storage
219:     // wrap, which also ends the transfer.
220:
221:     if(tempbcd.TestGMWM()) {
222:         IntEndofTransfer = true;
223:         EndofRecord = true;
224:     }
225:     else {
226:         if(MoveMode) {
227:             tempbcd.ClearWM();
228:             tempbcd.SetOddParity();
229:         }
230:         SetR1(tempbcd);
231:         if(CPU -> StorageWrapLatch) {
232:             IntEndofTransfer = true;
233:             EndofRecord = true;
234:         }
235:     }
236:
237:     // Now, if we have anything for the device, send it. We continue
238:     // doing this for up to two characters, in case R1 and R2 are both
239:     // full. We have to do it this way so that if, at the end, we set
240:     // ExtEndofTransfer, all of the data will have been sent.
241:
242:     while(GetR1Status() || GetR2Status()) {
243:         if(GetR1Status() & !GetR2Status()) {
244:             MoveR1R2();
245:         }
246:         CurrentDevice -> DoOutput();
247:         ResetR2();
248:     }
249:
250:     // Advance the appropriate address register to the next location.
251:     // (Storage wrap was detected earlier, as necessary!).
252:
253:     addr -> Set(CPU -> STARMOD(+1));
254:
255:     // Now that the data is all sent, if we are at IntEndofTransfer (either
256:     // From a GMWM or from storage wrap), also set ExtEndofTransfer so that
257:     // we quit. Note that the device may have already set ExtEndofTransfer!
258:
259:     if(IntEndofTransfer) {
260:         ExtEndofTransfer = true;
261:     }
262:
263: }
264:
```

```

265: // Channel Input Processing (shared overlap/not overlap code)
266:
267: void T1410Channel::DoInput(TAddressRegister *addr) {
268:
269:     CycleRequired = false;                                // Reset cycle required
270:     InputRequest = false;                               // Reset co-routine flag
271:
272:     if(!IntEndofTransfer) {                            // Skip this if wrapped!
273:         *(CPU -> STAR) = *addr;                      // Copy memory address
274:         CPU -> Readout();                           // Get existing memory
275:         CPU -> AChannel -> Select(                  // Gate A channel approp.
276:             this == CPU -> Channel[CHANNEL1] ?          TACchannel::A_Channel_E : TACchannel::A_Channel_F );
277:     }
278:
279:     // If no data from device, the end is near...
280:
281:     if(!(GetR1Status() || GetR2Status())) {
282:         LastInputCycle = true;
283:     }
284:
285:     // If B GMWM, we are typically done storing data (unless this
286:     // is read to end of core, which ignores GMWM). Unless, of course,
287:     // we have already hit end of record.
288:
289:     // Note that we can only check the op mod if we are NOT overlapped
290:     // (Note that the "to end of core" mods are all not overlapped)
291:
292:
293:     if(!EndofRecord && CPU -> B_Reg -> Get().TestGMWM()) {
294:         if(ChOverlap -> State() ||           (CPU -> Op_Mod_Reg -> Get().To6Bit()) != OP_MOD_SYMBOL_DOLLAR) {
295:             EndofRecord = true;
296:         }
297:     }
298:
299:
300:     // If no more input, or we hit GMWM,
301:     // set internal end of transfer, but keep on accepting characters
302:     // until External end of transfer.
303:
304:     if(LastInputCycle || EndofRecord) {
305:         IntEndofTransfer = true;
306:         if(ExtEndofTransfer) {
307:             return;
308:         }
309:     }
310:
311:     // If we get here, we are either ending, or we are still storing data
312:
313:     if(!LastInputCycle && !IntEndofTransfer) {        // Still storing input?
314:
315:         // If parity is no good, set data check...
316:
317:         if(!CPU -> AChannel -> Select().CheckParity()) {
318:             SetStatus(GetStatus() | IOCHDATACHECK);
319:
320:             // If asterisk insert, store an asterisk!
321:
322:             if(FI1415CE -> AsteriskInsert -> Checked) {
323:                 ChR2 -> Set(BCD_ASTERISK);
324:                 CPU -> AChannel -> Select(
325:                     this == CPU -> Channel[CHANNEL1] ?          TACchannel::A_Channel_E : TACchannel::A_Channel_F );
326:                 }
327:             } // End initial parity check
328:
329:             // Store the data (perhaps with a parity error!
330:
```

```
331:     CPU -> Store(CPU -> AssemblyChannel -> Select(
332:         (MoveMode ?
333:             TAssemblyChannel::AsmChannelWMB :
334:                 TAssemblyChannel::AsmChannelWMA),
335:             TAssemblyChannel::AsmChannelZonesA,
336:             false,
337:             TAssemblyChannel::AsmChannelSignNone,
338:             TAssemblyChannel::AsmChannelNumA) );
339:
340:
341: // If we have bad data, but not asterisk insert, STOP
342: // One good way: Run FORTRAN w/o Asterisk Insert! ;)
343:
344: if(!CPU -> AChannel -> Select().CheckParity()) {
345:     CPU -> AChannelCheck -> SetStop("Data Check, No Asterisk Insert!");
346:     return;
347: }
348:
349:     ResetR2();                                // Reset Channel Data
350:     if( CPU -> StorageWrapCheck(+1)) {        // If storage wrap -- done
351:         IntEndofTransfer = true;
352:         EndofRecord = true;
353:     }
354:     else {
355:         addr -> Set(CPU -> STARMod(1));      // Bump address register
356:     }
357:
358: } // End, !LastInputCycle
359:
360: // In the real world, the device keeps reading data, and will
361: // then set CycleRequired. (In fact, the console matrix will
362: // call ChannelStrobe to actually do this when a key is pressed).
363: // But most of our input devices in the emulator are co-routines,
364: // so we have to call them back to give them a chance to strobe
365: // the channel again. Also, in the real world, a device would
366: // keep reading input data after the channel set Internal End of
367: // Transfer -- we have to give the input co-routine a way to
368: // finish reading it's record, even if we are no longer storing
369: // it because we hit a GMWM or wrapped storage.
370:
371: CurrentDevice -> DoInput();
372:
373: }
374:
375: bool T1410Channel::ChannelStrobe(BCD ch) {
376:
377:     // If R1 has data already, move it to R2.
378:     // (In the emulator, that should probably never happen!)
379:
380:     if(GetR1Status()) {
381:         MoveR1R2();
382:     }
383:
384:     // Load R1, and copy to R2 if there is room in R2.
385:
386:     SetR1(ch);
387:     if(!GetR2Status()) {
388:         MoveR1R2();
389:     }
390:
391:     // If the channel has not already terminated the transfer,
392:     // ask to send the data to memory.
393:
394:     if(!IntEndofTransfer) {
395:         CycleRequired = true;
396:     }
```

```
397:     InputRequest = true;                                // Force co-routine call
398:     return true;
399:
400: }
401:
402: □
403:
404: // Class T1410IODevice implementation. This is an *abstract* base class,
405: // intended to be used to derive actual I/O devices
406:
407: T1410IODevice::T1410IODevice(int devicenumber, T1410Channel *Ch) {
408:     Channel = Ch;
409:     Ch -> AddIODevice(this,devicenumber);
410: }
411:
412: // And, finally, the 1410 IO Instruction Routines. We implement them
413: // here to keep the I/O stuff together!
414:
415: // Move and Load mode (M and L) Instructions.
416:
417: // Note: The channel selected by the CPU is known to be available, as the
418: // interlock test was passed during instruction readout at I3.
419: // IOChannelSelect indicates the selected channel.
420: // Channel -> ChUnitType has Device Type (e.g. 'T' for console)
421: // Channel -> ChUnitNumber has Unit Number
422:
423: void T1410CPU::InstructionIO() {
424:
425:     BCD opmod;
426:     T1410Channel *Ch = Channel[IOChannelSelect];
427:
428:     opmod = (Op_Mod_Reg -> Get()).To6Bit();
429:     assert(!(Ch -> ChInterlock -> State()));
430:
431:     // Reset the channel, then set Channel Interlock
432:
433:     Ch -> Reset();
434:     Ch -> ChInterlock -> Set();
435:
436:     // Set Move mode or Load mode, appropriately
437:
438:     if((Op_Reg -> Get()).To6Bit() == OP_IO_MOVE) {
439:         Ch -> MoveMode = true;
440:     }
441:     else {
442:         Ch -> LoadMode = true;
443:     }
444:
445:     // Start things out, depending on op modifier.
446:
447:     switch(opmod.ToInt()) {
448:
449:         case OP_MOD_SYMBOL_R:
450:         case OP_MOD_SYMBOL_DOLLAR:
451:             Ch -> ChRead -> Set();
452:             break;
453:
454:         case OP_MOD_SYMBOL_W:
455:         case OP_MOD_SYMBOL_X:
456:             Ch -> ChWrite -> Set();
457:             break;
458:
459:         default:
460:             InstructionCheck ->
461:                 SetStop("Instruction Check: Invalid I/O d-character");
462:             return;

```

```
463:     } // End switch on op modifier
464:
465:     // See if there is a device for this device number.  If not,
466:     // return not ready.
467:
468:     if(Ch -> GetCurrentDevice() == NULL) {
469:         Ch -> SetStatus(IOCHNOTREADY);
470:         IRingControl = true;
471:         return;
472:     }
473:
474:     // Start up the I/O, do initial status check (Status Sample A)
475:     // Just return if the status is not 0.
476:
477:     Ch -> SetStatus(Ch -> GetCurrentDevice() -> Select());
478:     if(Ch -> GetStatus() != 0) {
479:         IRingControl = true;
480:         return;
481:     }
482:
483:     // If OK so far, set Overlap/NotOverlap
484:
485:     if(CPU -> IOOverlapSelect) {
486:         Ch -> ChOverlap -> Set();
487:         // TODO: Ch -> RequestOutput() or RequestInput (Write/Read)
488:         IRingControl = true;
489:         return;
490:     }
491:     else {
492:         Ch -> ChNotOverlap -> Set();
493:     }
494:
495:     // If we get here, we are not overlapped.
496:
497:     if((opmod.ToInt() & OP_MOD_SYMBOL_W) == OP_MOD_SYMBOL_W) {
498:         CPU -> Display();
499:         while(!Ch -> ExtEndofTransfer) {
500:             Ch -> DoOutput(B_AR);
501:         }
502:         Ch -> ChNotOverlap -> Reset();
503:         Ch -> SetStatus(Ch -> GetCurrentDevice() -> StatusSample());
504:         IRingControl = true;
505:         return;
506:     }
507:     else if((opmod.ToInt() & OP_MOD_SYMBOL_R) == OP_MOD_SYMBOL_R) {
508:
509:         // Input processing continues so long as not External End from device
510:
511:         CPU -> Display();
512:         while(!Ch -> ExtEndofTransfer) {
513:
514:             // If no input, just wait here (not overlapped -- stuck here)
515:
516:             while(!(Ch -> CycleRequired || Ch -> InputRequest ||
517:                   Ch -> ExtEndofTransfer)) {
518:                 Application -> ProcessMessages();
519:                 // sleep(10);
520:                 continue;
521:             }
522:
523:             Ch -> DoInput(B_AR);
524:
525:         } // End, not External End of Transfer
526:
527:         assert(Ch -> ExtEndofTransfer);
528:
```

```
529:         // If, at the end, things do not match up ==> Wrong Length Record
530:         // (Note that we do *not* test InputRequest here!)
531:
532:         Ch -> SetStatus(Ch -> GetCurrentDevice() -> StatusSample());
533:         if(Ch -> CycleRequired || Ch -> GetR2Status() || !Ch -> EndofRecord) {
534:             Ch -> SetStatus(Ch -> GetStatus() | IOCHWLRECORD);
535:         }
536:
537:         // And all done - continue with Instructions
538:
539:         Ch -> ChNotOverlap -> Reset();
540:         IRingControl = true;
541:         return;
542:     }
543: }
544:
545:
```

```
    printf("\n\n");  
}
```

(1)

~~void~~ T1410CPU::InstructionUnitControl() {

    bcd opmod;

    T1410Channel \*ch = Channel[IOChannelSelect];

    opmod = (Op-Mod-Reg → Get().To 6 Bit());  
 assert (! (ch → chInterlock → State));

    ch → Reset();

    ch → chInterlock → Set();

    if (ch → GetCurrentDevice() == NULL) {

        ch → SetStatus (IOCHNOTREADY);

        IRingControl = true;

        return;

}

    ch → SetStatus (ch → GetCurrentDevice() → Select());

    if (ch → GetStatus() != 0) {

        IRingControl = true;

        return;

}

    if (CPU → IOoverlapdetet) {

        ch → ChOverlap → Set();

        // TO DO: Start unit op

        IRingControl = true;

        return;

}

    else {

        ch → ChNotOverlap → Set();

}

## Instruction Unit Control () cont

(2)

```
while (!Ch → ExtEnd of Transfer) {  
    Ch → DoUnitControl(opmod);  
}  
Ch → ChNotOverlap → Reset();  
Ch → SetStatus(Ch → GetCurrentDevice()) → StatusSample());  
IRigControl = true;  
return;
```

}

## TTape Unit

char filename[MAXPATH];  
FILE fd;  
bool irg;  
int unit;  
bool loaded;  
bool fileprotect;  
bool endof tape;  
char tapebuffer[ ];  
char tape-char;  
bool ready;  
bool selected;  
bool tapeindicate;  
bool highdensity;  
bool bot;  
~~bool mounted~~

TAPE-IRG      0x84  
TAPE-TM      0xF

## FTapeUnit

```
bool Selected() { return select; }
bool IsRead() { return ready; }
bool IsFileProtected() { return fileprotect; }
bool TapeIndicate() { return tapeindicate; }
bool HighDensity() { return highdensity; }
char *GetFileName() { return filename; }
```

Select (\*\*\*\*)

Reset (units) Reset () all units

Rewind

Rewind / Unload

Backspace

Read

Write (data)

Write () IRO

SIMUS 5/8

251-3913

39748524.50\$ - STILL Being processed

Prepare Forms

had it wrong! (DUT FCIN)

6.5.8 - for 1e<sup>2</sup>

6.10

dj125en.

## Tape Unit

`TapeUnit(int u);` constructor

④ `bool Select(bool b)`

Init()

⑤ `bool Rewind()`

⑥ `bool RewindUnload()`

⑦ `bool BackSpace()` -4 internal error -3 not ready >= 0 char

⑧ `int Read()`

⑨ `bool Write(char c)`

⑩ `void WriteIRG()`

⑪ ~~bool~~ WriteTM()

⑫ `bool Skip()`

⑬ `bool LoadRewind(char *filename)`

⑭ `bool Reset()`

⑮ `bool Unload()`

⑯ `bool Start()`

⑰ `bool Mount()`

T Tape Unit:: U1729 Tape Unit (int u);

INTC()

Unit = u;

fd = NULL;

irg = loaded = file protect = end of type = false;

filename [0] = '10';

selected = false;

high density = true;

dot = false;

```
bool TTypeUnit::Reset()
```

```
    ready = false;  
    return (true);
```

```
}
```

```
bool TTypeUnit::LoadRewind()
```

```
if (ready) {  
    return (false);
```

```
}
```

```
    }
```

```
if (fd != NULL) {  
    if (lseek(fd, 0L, -1) == -1) {  
        fd = NULL;  
        return (false);
```

```
}
```

```
    lrg = endoftape = tapeindicate = false;
```

```
    return (bot = true);
```

```
}
```

```
    loaded = true;
```

```
bool TTapeUnit::Unload() {  
    if (ready || loaded) {  
        return (false);  
    }  
    if (fd != NULL) {  
        fclose(fd);  
        fd = NULL;  
    }  
    irg = loaded = fileprotect = endoffile = ready = false;  
    tapeindicate = false;  
    return (true);  
}
```

```
bool TTapeUnit::Mount(char *f) {  
    if (ready || loaded) {  
        return (false);  
    }  
    assert(strlen(f) + 1 < sizeof filename);  
    assert(fd == NULL);  
    strcpy(filename, f);  
    irg = loaded = fileprotect = endoffile = tapeindicate = false;  
    return (true)  
}
```

```
bool JTapeUnit::start() {
    if (ready || !loaded) {
        return (false);
    }

    if (fd == NULL) {
        if ((fd = fopen(filename, "rb+")) == NULL) {
            if ((fd = fopen(filename, "rb")) == NULL) {
                ready = false;
                loaded = false;
            } else {
                DEBUG("start(): open unit file %s", unit);
                DEBUG(perror(errno), errno);
                fileprotect = true;
                active = true;
            }
        }
    }

    else {
        fileprotect = false;
        ready = true;
    }
}

else {
    ready = true;
}

return (ready);
}
```

To  
implement

```
bool TTapeUnit::Select(bool b)
```

```
    return selected = b;
```

```
}
```

```
bool UTapeUnit::Rewind()
```

```
{ if (!selected || !loaded || !ready) {
```

```
    return (false);
```

```
}
```

```
assert (fd != NULL);
```

```
if (lseek(fd, 0) ==   ) {
```

```
    fd = NULL;
```

```
    ready = false;
```

```
}
```

```
    loaded = false;
```

```
    return (true);
```

```
}
```

```
    tapeIndicator = false;
```

```
    return (b = true);
```

```
}
```

```
bool TTapeUnit::RewindUnload() {
    if (!Rewind()) {
        return (false);
    }
    if (fd != NULL) {
        fclose(fd);
        fd = NULL;
    }
    irg = loaded = file protect = endof tape = ready = false;
    tape indicate = false;
    return (true);
}

bool TTapeUnit::Skip() {
    return (true);
}
```

①

```

bool TTopcUnit::BackSpace() {
    if (!selected || !ready || !loaded) {
        return (false);
    }
    if (bot) {
        tapeindicator = true;
        return (true);
    }
    while (true) {
        if (fseek(fd, -2L, SEEK_CUR) != 0) {
            DEBUG("Tape unit backspace Error", errno);
            ready = false; ← loaded = false;
            fclose(fd);
            return (false);
        }
        if (ftell(fd) == 0L) {
            iog = false;
            bot = true;
            return (true);
        }
        if (fread(&tape-buffer, 1, 1, fd) != 1) {
            if (!feof(fd)) {
                DEBUG("Tape backspace error (read)", errno);
                fclose(fd); ← loaded = false;
                ready = false;
                return (false);
            }
        }
    }
    return (true);
}

```

DEBUG  
return (false);

... while (true)

```
if (tapebuffer & TAPE-IRG){  
    tapebuffer &= (~TAPE-IRG);  
    irg = true;  
    return (true);
```

{

{

```

bool TTapeUnit::Write (char c) {
    if (irg) {
        c |= TAPE_IRG;
    }
    if (!loaded || !ready || fd == NULL) {
        DEBUG("write(): Tape unit not ready, ", unit);
        return (false);
    }
    if (fwrite (&c, 1, 1, fd) != 1) {
        DEBUG("write(): File I/O error writing unit ", unit);
        DEBUG(perror (errno), 0);
        tapeindicate = true;
        ready = false;
        fclose (fd);
        loaded = false;
        return (false);
    }
    irg = false;
    return (true);
}

```

```

void TTapeUnit::WriteIRG() {
    irg = true;
    return;
}

bool TTapeUnit::WriteTM() {
    return (write (BD_TM | TAPE_IRG));
}

```

```
int TTape Unit:: Read() {
    int rc;
    if (!loaded || !ready || fd == NULL) {
        DEBUG ("Tape Unit Not Ready", unit);
        return (-3);
    }
    if ((c = ReadNextChar()) < 0) {
        return (tape-buffer);
    }
    if (irg && (tape-buffer & 0x3f) == BCD-TM)) {
        if (crc = ReadNextChar()) < 0) {
            return (rc);
        }
        return (-2);
    }
    if (bot || irg) {
        tapebuffer &= (~TAPE-IRG);
        bot = irg = false;
    }
    if (tapebuffer & TAPE-IRG) {
        irg = true;
        return (-1);
    }
    return (tape-buffer);
}
```

```
int TTapeUnit :: ReadNextChar() {
    char c
    if (fread(&c, 1, 1, fd) != 1) {
        if (!feof(fd)) {
            DEBUG("File I/O Error reading tape ", errno);
            close(fd);
            ready = false;
            loaded = false;
            return (-4);
        }
        tapeindicate = true;
        return (-2);
    }
    return(c);
}
```

Channel →

Add IODriver (this, **B**)

Channel Strobe (bcd)

input char (asm, parity ~~etc~~)

INQ RISE

External Transfer

VERBOSITY

UI14I0ARDTH.H  
UI14I0DEBUG.H  
UI14I0DATA.H  
UI14I0BRANCH.H  
UF14I0MEJC.H  
→ UI14I0CHANNEL.H

✓ UI14I4.H ✓ UI14I0I.CPP ✓ UI14I0A.H  
✓ UI14I0CAUT.H (needs UBOD.H)

✓ UI14I0INST.H

OK UI14I0CPU1.CPP  
✓ UI14I0INST.CPP  
✓ UI14I0ARDTH.CPP

DATA.CPP

BRANCH

MSG

DEBO

UBOD.H ←  
UI14I0CPU1.H OK

UI14I0DEBUG.H CHANNEL  
Lassert.H ~~etc~~

✓ UI14ISCE.H  
✓ UI14ISIO.H (needs UBOD.H  
UI14I0CPU1.H)

✓ UI14ISIO.H

OK PISM14I0.CPP  
OK UI14ISIO.CPP  
OK PWR.CPP  
CICE  
OK UI14I0INST.CPP  
OK CHNL

✓ UI14I0L.H  
UBOD.H

✓ UI14I5L.CPP  
OK UI14I0CPU1.H  
✓ UI14I0INST.CPP

UBOD.H ←  
UI14I0CPU1.H

✓ UI14I0CPU1.H  
UI14I0CPU1.H (needs UBOD.H)  
UI14I0DEBUG.H

✓ UI14I0CPU1.H  
UBOD.H)

✓ UI14I0PWR.H ✓ PISM14I0.CPP ✓ UI14I0PWR.CPP  
✓ UI14I0CPU1.H  
UI14I0I.H  
UI14ISIO.H (needs UBOD.H)

✓ DEBUG → stdio

```

516: class T1410IODevice : public TObject {
517:
518: protected:
519:     T1410Channel *Channel;           // Channel device is attached to
520:
521: public:
522:     T1410IODevice(int devicenum, T1410Channel *Channel); // Constructor
523:     virtual int Select() = 0;          // Start an operation
524:     virtual int StatusSample() = 0;    // Return device status
525:     virtual void DoOutput() = 0;       // Channel -> Device
526:     virtual void DoInput() = 0;        // Device -> Channel
527: };

```

~~Virtual~~ void DoUnitControl(bcd dcher) {
 Channel → SetStatus( ... );
 Channel → GetStatus() | IOCHNOTREADY;
 DEBUG("Unimplemented Unit Control, Unit ", 0);
 return;
}

IO Unit  
~~Channel~~ ~~DoUnitControl~~ ~~Unit~~  
 U1

~~class~~  
~~T1410Channel~~

C ATALOG# — why not a ~~global~~ ~~global~~ list, field too small

X Delete Didn't work

Devour + Contains No Data

void T1410Channel::DoUnitControl( Dcd opmod ) {

    Current Device → DoUnitControl( opmod );

    return;

}

```
//  
// A little "hack" for printing BCD tapes, with special options for 1400 series  
// systems that used word separator characters for word marks (1401, 1410)  
// Output is translated from ASCII to BCD. Word Marks are *NOT* indicated.  
//  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <ctype.h>  
#include <alloc.h>  
  
#include "UBCD.H"  
  
#define TAPE_IRG 0200  
#define TAPE_C_BIT 0100  
  
#define STORAGE_WM_BIT 0x80  
  
#define BCD_TM 017  
#define BCD_WS 035  
#define BCD_AS 020  
  
#define TAPE_BUFFER_SIZE 100000L  
  
char load_mode = 0; // 1 to process word separators  
char load_mode_1410 = 0; // 1 to process word separators as a 1410 would  
char parity = 0; // 0 for Even, 1 for ODD  
char verbose = 1; // 1 for verbose mode (default), 0 for "quiet" mode  
int print_len = 72; // How much data to print, per record  
long print_start = 0; // Offset to start printing at  
long record_start = 1; // Assume we start with first record  
int record_count = -1; // Assume we process all records  
long core_image = -1; // Assume we are not creating a core image file  
FILE *fd_image = NULL;  
  
//  
// Parity table for the 64 BCD characters (without parity bit,  
// wordmark bits, etc).  
//  
  
int parity_table[64] = {  
    0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,  
    1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0  
};  
  
int main(int argc, char **argv)  
{  
    FILE *tape;  
  
    unsigned char far *tape_buffer;  
    unsigned char far *tape_char;  
  
    long record = 0;  
    int eof = 0;  
  
    long offset = 0;  
    long mem_offset;  
    long l;  
  
    int wm = 0;  
    int at_irg = 0;  
    long image_out;
```

```
unsigned char bcd_char,junk;

void usage();

int tape_parity(unsigned char c);
void do_print(unsigned char *buffer,long offset,int print_len);

// Do options processing

while (--argc && **(++argv) == '-') {
    switch (tolower((*argv)[1])) {
    case 'l':
        load_mode = 1;
        load_mode_1410 = 0;
        break;
    case 't':
        load_mode = 1;
        load_mode_1410 = 1;
    case 'o':
        parity = 1;      // Odd parity
        break;
    case 'p':
        print_len = atoi(&(*argv)[2]);
        break;
    case 's':
        print_start = atoi(&(*argv)[2]);
        break;
    case 'q':
        verbose = 0;
        break;
    case 'r':
        record_start = atoi(&(*argv)[2]);
        break;
    case 'c':
        record_count = atoi(&(*argv)[2]);
        break;
    case 'i':
        core_image = atoi(&(*argv)[2]);
        if ((fd_image = fopen("image.cor","wb")) == NULL) {
            fprintf(stderr,"Cannot open/create file image.cor\n");
            exit(2);
        }
        record_count = 1;
        break;
    default :
        fprintf(stderr,"Unknown option: %s\n",*argv);
        usage();
    }
}

if(argc != 1) {
    usage();
}

if(verbose) {
    printf("Tape file name is %s\n",*argv);
    printf("Print length per record is %d starting at offset %d\n",
           print_len,print_start);
    printf("Starting at record %ld for %d records\n",record_start,record_count);
}

if((tape = fopen(*argv,"rb")) == NULL) {
```

```
fprintf(stderr,"Can't open tape file %s: ",*argv);
perror("");
exit(1);
}

if((tape_buffer = (unsigned char far *)farcalloc(TAPE_BUFFER_SIZE,1)) == NULL) {
    fprintf(stderr,"farcalloc of tape buffer failed...\n");
    exit(1);
}

if(fread(&bcd_char,1,1,tape) != 1) {
    perror("C I/O error reading first tape char: ");
    free(tape_buffer);
    exit(1);
}

// Set up to be at an IRG initially
at_irg = 1;

// This is the main loop for processing the tape - character by character
while (!eof) {

    // Handle end of record processing (including first time thru)

    if(at_irg) {
        at_irg = 0;                                // reset EOR indicator (esp. 1st time)
        ++record;                                 // bump record number
        offset = 0;                               // reset offset in record
        mem_offset = 0;                           // reset offset in memory
        tape_char = tape_buffer;                  // prime the bump
        bcd_char &= (~TAPE_IRG);                 // check if next character is EOR too

        // Handle tape mark processing (always at beginning of record)

        if(bcd_char == BCD_TM) {
            printf("Tape mark...\n");
            //
            // Throw away (0x0F) character after tape mark -- garbage
            //
            if(fread(&junk,1,1,tape) != 1) {
                perror("C I/O error reading char after tape mark: ");
                exit(1);
            }
        }
    }

    // Process ordinary characters (not at end of record last time thru)

    else {
        if(fread(&bcd_char,1,1,tape) != 1) {
            if(feof(tape)) {
                eof = 1;
            }
            else {
                perror("C I/O error reading tape char: ");
                exit(1);
            }
        }
    }

    // Check to see if we are at a new end of record. Handle accordingly
}
```

```
// (For example, we print when we reach end of record).

at_irg = bcd_char & TAPE_IRG;
if(at_irg && load_mode && wm) {
    printf("** Trailing word separator in load mode.\n");
}

if(at_irg) {

    // Do the printing. This could be replaced by a special
    // analysis routine, if you like

    if(record >= record_start &&
       (record_count == -1 || record < record_start+record_count)) {
        if(verbose) {
            printf("Record %ld: length %ld",record,offset);
            if(load_mode) {
                printf(", length in memory: %ld",mem_offset);
            }
            printf("\n");
        }

        // If we are saving an image file, do it at first printed record

        if(core_image > -1) {
            printf("Saving core image...");
            image_out = BITC;
            if(fprintf(fd_image,"%05d",mem_offset+core_image) != 5) {
                fprintf(stderr,"Error writing image size.\n");
                fclose(fd_image);
                exit(2);
            }
            for(l=0; l < core_image; ++l) {
                if(fwrite(&image_out,sizeof (long),1,fd_image) != 1) {
                    fprintf(stderr,"Error writing image C bit fill leader.\n");
                    fclose(fd_image);
                    exit(2);
                }
            }
            for(l=0; l < mem_offset; ++l) {
                image_out = tape_buffer[l];
                if(fwrite(&image_out,sizeof (long),1,fd_image) != 1) {
                    fprintf(stderr,"Error writing core image data.\n");
                    fclose(fd_image);
                    exit(2);
                }
            }
            core_image = 0; // Do this only once
        } // End write core image

        do_print(tape_buffer,print_start,
                 (print_len < mem_offset-print_start+1) ?
                 print_len : mem_offset-print_start+1 );

    }
    continue;
}

// Check for parity errors, and notify, then strip the parity bit

if(tape_parity(bcd_char) != parity) {
```

```
    printf("## Parity error record %ld offset %ld\n",record,offset);
}

++offset;

// Check for word separator. If we found one, and we are in load
// mode, don't store it, just remember it was here

// A 1401 (apparently) took any number of word separator characters,
// and turned them into a trailing wordmark. The 1410, on the other
// hand, would turn two consecutive word separator characters into a
// word separator character in memory.

if(load_mode && ((bcd_char & 0x3f) == BCD_WS)) {

    if(!load_mode_1410) {                                // 1401 mode
        wm = 1;
    }
    else {                                              // 1410 mode
        ++wm;
        if(wm == 2) {
            wm = 0;                                     // Process 2nd WS normally
        }
    }

    if(wm) {
        continue;                                       // If wm bit set, skip to next
    }
}

// Store character into buffer

++mem_offset;
if(mem_offset >= TAPE_BUFFER_SIZE) {
    fprintf(stderr,"Internal error: offset >= buffer size.\n");
    exit(1);
}

*tape_char = bcd_char;

// If we are remembering a word separator, add the WM bit, and flip
// the check bit.

if(wm) {
    *tape_char |= STORAGE_WM_BIT;
    *tape_char ^= TAPE_C_BIT;
}

// Prepare for next character

++tape_char;
wm = 0;

}

printf("EOF after %d records\n",record);

free(tape_buffer);
fclose(tape);
printf("Done.\n");
return (0);
}
```

```
void usage()
{
    fprintf(stderr,"Usage: bcdtape [-i] [-l|-t] [-o] [-p#] [-s#] [-r#] [-c#] <tapefile>\n");
    fprintf(stderr," -l: Process word separators as a 1401 would (load mode)\n";
    fprintf(stderr," -t: Process word separators as a 1410 would (load mode)\n";
    fprintf(stderr," -o: Process tape in odd parity\n");
    fprintf(stderr," -i#: Create core image file as though loading at # (implies -c1)\n");
    fprintf(stderr," -s#: Start printing record at offset #\n");
    fprintf(stderr," -p#: Print # characters per record\n");
    fprintf(stderr," -r#: Start printing at record #\n");
    fprintf(stderr," -c#: Print # records\n");
    exit(1);
}

int tape_parity(unsigned char c)
{
    int bits = 0;
    unsigned char bit;

    if(TAPE_C_BIT &c) {
        bits=1;
    }

    bits += parity_table[c & 077];

    return (bits % 2);
}

// Analysis / print routine

void do_print(unsigned char *tape_buffer,long print_start,int print_len)
{
    long i,j;
    long print_end,line_end;

    // If we are in load mode, show the word marks, too.

    print_end = print_start+print_len;

    for(i=print_start; i < print_start+print_len; i += 80) {

        line_end = (print_end) > i+80 ? i+80 : print_end;

        // If load mode, print wordmarks as "v"

        if(load_mode) {
            for(j = i; j < line_end; ++j) {
                printf("%c", (tape_buffer[j] & STORAGE_WM_BIT) ? 'v' : ' ');
            }
            printf("\n");
        }

        // Print the record itself, in ASCII, stripping the word-mark bit

        for(j=i; j < line_end; ++j) {
            printf("%c",bcd_ascii[tape_buffer[j] & 0x3f] & 0x7f);
        }

        printf("\n");
    }
}
```

```
1: //-----
2: #ifndef UI1410CPUTH
3: #define UI1410CPUTH
4:
5: #include "ubcd.h"
6:
7: typedef void (_closure *TInstructionExecuteRoutine)();
8:
9: extern long ten_thousands[], thousands[], hundreds[], tens[];
10: extern long scan_mod[];
11: extern unsigned char sign_normalize_table[];
12: extern unsigned char sign_complement_table[];
13: extern unsigned char sign_negative_table[];
14:
15: struct OpCodeCommonLines {
16:     unsigned short ReadOut;
17:     unsigned short Operational;
18:     unsigned short Control;
19: };
20:
21:
22: extern struct OpCodeCommonLines OpCodeTable[64];
23:
24: extern int IndexRegisterLookup [];
25:
26: // Table of execute routines. These have to be closures to inherit
27: // the object pointer (CPU). Closures have to be initialized at runtime.
28: // We do it in the constructor.
29:
30: extern TInstructionExecuteRoutine InstructionExecuteRoutine[64];
31:
32: //
33: // Classes (types) used to implement the emulator, including the
34: // final class defining what is in the CPU, T1410CPU.
35:
36: //
37: // Abstract class designed to build lists of objects affected by Program
38: // Reset and Computer Reset
39: //
40:
41: class TCpuObject : public TObject {
42:
43: public:
44:     TCpuObject();                                // Constructor to init data
45:     virtual void OnComputerReset() = 0;           // Called during Computer Reset
46:     virtual void OnProgramReset() = 0;             // Called during Program Reset
47:
48: protected:
49:     bool DoesProgramReset;                      // true if this is reset by P.R. button
50:
51: public:
52:     TCpuObject *NextReset;
53: };
54:
55:
56: //
57: // A second abstract class of objects that not only react to the Resets,
58: // but also have entries on the display panel.
59: //
60:
61: class TDisplayObject : public TCpuObject {
62:
63: public:
64:     TDisplayObject();                            // Constructor to init data.
65:     virtual void Display() = 0;                  // Called to display this item
66:     virtual void LampTest(bool b) = 0;            // Called to start/end lamp test
```

```
67:
68: public:
69:     TDisplayObject *NextDisplay;
70: };
71:
72: □
73:
74: // Class TDisplayObjects are indicators: They just
75: // display other things. As a result, they need a pointer to
76: // a function returning bool in order to decide what to do.
77:
78: class TDisplayIndicator : public TDisplayObject {
79:
80: protected:
81:     TLabel *lamp;
82:     bool (_closure *display)();
83:
84: public:
85:
86:     // The constructor requires a pointer to a lamp and a pointer to
87:     // a function that can calculate lamp state.
88:
89:     // We use a closure so that we don't have to pass a pointer to the
90:     // stuff the display function needs (an object pointer is automatically
91:     // embedded in an __closure *)
92:
93:     TDisplayIndicator(TLabel *l,bool (_closure *func)()) {
94:         lamp = l;
95:         display = func;
96:     }
97:
98:     virtual void OnComputerReset() { ; }      // These have no state to reset
99:     virtual void OnProgramReset() { ; }        // These have no state to reset
100:
101:    void Display() {
102:        lamp -> Enabled = display();
103:        lamp -> Repaint();
104:    }
105:
106:    void LampTest(bool b) {
107:        lamp -> Enabled = (b ? true : display());
108:        lamp -> Repaint();
109:    }
110: };
111:
112: □
113:
114: //
115: // Class of TDisplayObjects that are latches:
116: // that can be set, reset and their state read out. Some are
117: // reset by Program Reset (PR) some are not.
118: //
119:
120: class TDisplayLatch : public TDisplayObject {
121:
122: protected:
123:     bool state;                                // Latches can be set or reset
124:     bool doprogramreset;                        // Some are reset by PR, some are not.
125:     TLabel *lamp;                             // Pointer to display lamp.
126:
127: public:
128:     TDisplayLatch(TLabel *l);                  // Constructor - Set up lamp
129:     TDisplayLatch(TLabel *l, bool progreset); // Same, but inhibit PR
130:
131:     virtual void OnComputerReset();           // Define Computer Reset behavior now
132:     virtual void OnProgramReset();            // Define Program Reset behavior now too
```

```
133: 
134:     void Display();           // Define display behavior
135:     void LampTest(bool b);   // Define lamp test behavior.
136: 
137:     // All you can really do with latches is set/reset/test them
138: 
139:     inline void Reset() { state = false; }
140:     inline void Set() { state = true; }
141:     inline void Set(bool b) { state = b; }
142:     void SetStop(char *msg);
143:     inline bool State() { return state; }
144: };
145: 
146: □
147: 
148: class TRingCounter : public TDisplayObject {
149: private:
150:     char state;           // Override "state" variable !!
151:     char max;             // Max number of entries
152:     TLabel *lastlamp;    // Last lamp to be displayed
153:     TLabel *lastlampCE;  // Last CE lamp to be displayed
154: 
155: public:
156:     TLabel **lamps;      // Ptr to array of lamps
157:     TLabel **lampsCE;    // Ptr to array of CE lamps (or 0)
158: 
159: public:
160:     TRingCounter(char n); // Construct with # of entries
161:                         // Ring counters are always reset by PR
162: 
163:     virtual __fastcall ~TRingCounter(); // Destructor for array of lamps
164: 
165:     // Functions inherited from abstract base classes now need definition
166: 
167:     void OnComputerReset();
168:     void OnProgramReset();
169:     void Display();
170:     void LampTest(bool b);
171: 
172:     // The real meat of the Ring Counter class
173: 
174:     inline void Reset() { state = 0; }
175:     inline char Set(char n) { return state = n; }
176:     inline char State() { return state; }
177:     char Next();
178: };
179: 
180: □
181: 
182: // Data Registers. All are stored as BCD, but we have special
183: // set routines to set or clear special parts for those registers
184: // that don't use all the bits (e.g. the Op register has no WM or
185: // C bits.
186: 
187: // Also, a Register can have an optional pointer to an error latch.
188: // If so, then whenever the register is used, a parity check for ODD
189: // parity is made, and if invalid, the error latch is set. (This
190: // includes use during assignment).
191: 
192: class TRegister : public TDisplayObject {
193: 
194: private:
195: 
196:     BCD value;
197:     TLabel *lampER;      // If set, there is an error lamp
198:     TLabel **lamps;      // If set, they point to lamps for WM C B A 8 4 2 1
```

```
199:                         // Any lamp not present MUST be set to 0
200:
201: public:
202:
203:     TRegister() { value = BITC; DoesProgramReset = true; lampER = 0; lamps = 0; }
204:     TRegister(bool b) { value = BITC; DoesProgramReset = b; lampER = 0; lamps = 0; }
205:     TRegister(int i) { value = i; DoesProgramReset = true; lampER = 0; lamps = 0; }
206:     TRegister(int i, bool b) { value = i; DoesProgramReset = b; lampER = 0; lamps =
207:     0; }
208:     inline void OnComputerReset() { Reset(); }
209:     inline void Reset() { value = BITC; }
210:
211:     inline void Set(BCD bcd) { value = bcd; }
212:
213:     inline BCD Get() { return value; }
214:
215:     void operator=(TRegister &source);
216:     void Display();
217:     void LampTest(bool b);
218:
219:     // To set up a register to display, provide a pointer to an error
220:     // lamp (if any), and an array of pointers to data lamps. Data lamps
221:     // for any given bit (e.g. WM) may or may not exist. Order of lamps
222:     // is 1 2 4 8 A B C WM (0 thru 7)
223:
224:     void SetDisplay(TLabel *ler, TLabel **l) {
225:         lampER = ler;
226:         lamps = l;
227:     }
228:
229:     void OnProgramReset() {
230:         if(DoesProgramReset) {
231:             Reset();
232:         }
233:     }
234: };
235: □
236: // Address Registers. For efficiency, we keep both binary and
237: // the real 2-out-of-5 code representations. If either one is
238: // valid, and the other representation is requested, we convert
239: // on the fly (and mark that representation valid).
240:
241: // The Set() function effectively implements the Address Channel.
242:
243: class TAddressRegister : public TCpuObject {
244:
245: private:
246:
247:     long i_value;           // Integer equivalent of register
248:     bool i_valid;          // True if integer rep. is valid
249:     bool set[5];            // True if corresponding digit is set
250:     TWOOF5 digits[5];      // Original 2 out of 5 code representation
251:     bool d_valid;          // True if digit rep. is valid
252:     char *name;
253:
254: public:
255:
256:     void OnComputerReset() { Reset(); };
257:     void OnProgramReset() { };
258:
259:     TAddressRegister();    // Constructor / initialization
260:     bool IsValid();        // Returns true if all digits set
261:     long Gate();            // Returns integer value if valid, -1 if not.
262:     BCD GateBCD(int i);    // Returns a single digit
263:     void Set(TWOOF5 digit,int index); // Sets a digit
```

```
264:     void Set(long value);    // Sets whole register from binary (address mod)
265:     void Reset();           // Resets the register to blanks
266:
267:     void operator=(TAddressRegister &source); // Assignment
268:
269: };
270:
271: □
272:
273: // This class defines the A Channel - basically, it defines what
274: // register is selected when the A channel is accessed.
275:
276: class TAChannel : public TDisplayObject {
277:
278: public:
279:
280:     enum AChannelSelect { A_Channel_None = 0, A_Channel_A = 1,
281:                           A_Channel_Mod = 2, A_Channel_E = 3, A_Channel_F = 4 };
282:
283: private:
284:
285:     enum AChannelSelect AChannelSelect;
286:     TLabel *lamps[4];
287:
288: public:
289:
290:     TAChannel();                      // Constructor
291:
292:     void OnComputerReset() {
293:         Reset();
294:     }
295:
296:     void OnProgramReset() {
297:         Reset();
298:     }
299:
300:     inline BCD Select(enum AChannelSelect sel); // Select input to A Channel
301:     inline BCD Select();                         // Use whatever was last selected
302:
303:     inline void Reset() { AChannelSelect = A_Channel_None; }
304:
305:     void Display();
306:     void LampTest(bool b);
307: };
308:
309: □
310:
311: // The Assembly Channel: The 1410 Mix-Master!
312:
313: class TAssemblyChannel : public TDisplayObject {
314:
315: public:
316:
317:     enum AsmChannelZonesSelect {
318:         AsmChannelZonesNone = 0, AsmChannelZonesB = 1, AsmChannelZonesA = 2
319:     };
320:
321:     enum AsmChannelWMSelect {
322:         AsmChannelWMNone = 0, AsmChannelWMB = 1, AsmChannelWMA = 2,
323:         AsmChannelWMSet = 4
324:     };
325:
326:     enum AsmChannelNumericSelect {
327:         AsmChannelNumNone = 0, AsmChannelNumB = 1, AsmChannelNumA = 2,
328:         AsmChannelNumAdder = 3, AsmChannelNumZero = 4
329:     };

```

```
330:
331:     enum AsmChannelSignSelect {
332:         AsmChannelSignNone, AsmChannelSignB = 1, AsmChannelSignA = 2,
333:         AsmChannelSignLatch = 3
334:     };
335:
336: private:
337:
338:     BCD value;           // We remember value, for efficiency
339:     bool valid;          // True if value is set. Reset each cycle!
340:
341:     enum AsmChannelZonesSelect AsmChannelZonesSelect;
342:     enum AsmChannelWMSelect AsmChannelWMSelect;
343:     enum AsmChannelNumericSelect AsmChannelNumericSelect;
344:     enum AsmChannelSignSelect AsmChannelSignSelect;
345:     bool AsmChannelInvertSign;
346:
347:     bool AsmChannelCharSet; // True if value set to a special char
348:
349:     TLabel *AssmLamps[8];
350:     TLabel *AssmComplLamps[8];
351:     TLabel *AssmERLamp;
352:
353: public:
354:
355:     TAssemblyChannel();
356:
357:     void OnComputerReset() { Reset(); }
358:     void OnProgramReset() { Reset(); }
359:
360:     void Display();
361:     void LampTest(bool b);
362:
363:     BCD Select();           // Uses last value and state
364:
365:     BCD Select(
366:         enum AsmChannelWMSelect WMSelect,
367:         enum AsmChannelZonesSelect ZoneSelect,
368:         bool InvertSign,
369:         enum AsmChannelSignSelect SignSelect,
370:         enum AsmChannelNumericSelect NumSelect
371:     );
372:
373:     BCD Get();
374:
375:     BCD Set(BCD v) { value = v; valid = true; AsmChannelCharSet = true; }
376:
377:     void Reset();
378:
379:     bool SpecialChar() { return AsmChannelCharSet; };
380:
381:     BCD GateAChannelToAssembly(BCD v);
382:
383:     BCD GateBRegToAssembly(BCD v);
384:
385: };
386:
387: □
388:
389: // This class defines what is in an I/O Channel
390:
391: #define IOCHNOTREADY    1
392: #define IOCHBUSY        2
393: #define IOCHDATACHECK   4
394: #define IOCHCONDITION   8
395: #define IOCHNOTTRANSFER 16
```

```
396: #define IOCHWLRECORD      32
397:
398: #define IOLAMPNOTREADY    0
399: #define IOLAMPBUSY        1
400: #define IOLAMPDATACHECK   2
401: #define IOLAMPCONDITION   3
402: #define IOLAMPNOTTRANSFER  4
403: #define IOLAMPWLRECORD    5
404:
405: class T1410Channel : public TDisplayObject {
406:
407: public:
408:
409:     // Functions inherited from abstract base class classes now need definition
410:
411:     void OnComputerReset();
412:     void OnProgramReset();
413:     void Display();
414:     void LampTest(bool b);
415:
416: private:
417:
418:     // Channel information
419:
420:     int ChStatus;                      // Channel status (see defines)
421:
422:     TLabel *ChStatusDisplay[6];         // Channel status lights
423:
424: public:
425:
426:     TRegister *ChOp;
427:     TRegister *ChUnitType;
428:     TRegister *ChUnitNumber;
429:     TRegister *ChR1, *ChR2;
430:
431:     TDisplayLatch *ChInterlock;
432:     TDisplayLatch *ChRBCTInterlock;
433:     TDisplayLatch *ChRead;
434:     TDisplayLatch *ChWrite;
435:     TDisplayLatch *ChOverlap;
436:     TDisplayLatch *ChNotOverlap;
437:
438:     enum TapeDensity {
439:         DENSITY_200_556 = 0, DENSITY_200_800 = 1, DENSITY_556_800 = 2
440:     } TapeDensity;
441:
442:
443:     // Methods
444:
445:     T1410Channel(                                // Constructor
446:         TLabel *LampInterlock,
447:         TLabel *LampRBCTInterlock,
448:         TLabel *LampRead,
449:         TLabel *LampWWrite,
450:         TLabel *LampOverlap,
451:         TLabel *LampNotOverlap,
452:         TLabel *LampNotRead,
453:         TLabel *LampBusy,
454:         TLabel *LampDataCheck,
455:         TLabel *LampCondition,
456:         TLabel *LampWLRecord,
457:         TLabel *LampNoTransfer
458:     );
459:
460:     inline int SetStatus(int i) { return ChStatus = i; }
461:     inline int GetStatus() { return ChStatus; }
```

```
462: };  
463:  
464:  
465: □  
466: // This class defines what is actually inside the CPU.  
467:  
468: #define MAXCHANNEL 2  
469: #define CHANNEL1 0  
470: #define CHANNEL2 1  
471:  
472: #define STORAGE 80000  
473:  
474: #define I_RING_OP 0  
475: #define I_RING_1 1  
476: #define I_RING_2 2  
477: #define I_RING_3 3  
478: #define I_RING_4 4  
479: #define I_RING_5 5  
480: #define I_RING_6 6  
481: #define I_RING_7 7  
482: #define I_RING_8 8  
483: #define I_RING_9 9  
484: #define I_RING_10 10  
485: #define I_RING_11 11  
486: #define I_RING_12 12  
487:  
488: #define A_RING_1 0  
489: #define A_RING_2 1  
490: #define A_RING_3 2  
491: #define A_RING_4 3  
492: #define A_RING_5 4  
493: #define A_RING_6 5  
494:  
495: #define CLOCK_A 0  
496: #define CLOCK_B 1  
497: #define CLOCK_C 2  
498: #define CLOCK_D 3  
499: #define CLOCK_E 4  
500: #define CLOCK_F 5  
501: #define CLOCK_G 6  
502: #define CLOCK_H 7  
503: #define CLOCK_J 8  
504: #define CLOCK_K 9  
505:  
506: #define SCAN_N 0  
507: #define SCAN_1 1  
508: #define SCAN_2 2  
509: #define SCAN_3 3  
510:  
511: #define SUB_SCAN_NONE 0  
512: #define SUB_SCAN_U 1  
513: #define SUB_SCAN_B 2  
514: #define SUB_SCAN_E 3  
515: #define SUB_SCAN_MQ 4  
516:  
517: #define CYCLE_A 0  
518: #define CYCLE_B 1  
519: #define CYCLE_C 2  
520: #define CYCLE_D 3  
521: #define CYCLE_E 4  
522: #define CYCLE_F 5  
523: #define CYCLE_I 6  
524: #define CYCLE_X 7  
525:  
526: class T1410CPU {  
527:
```

```

528: private:
529:
530:     BCD core[STORAGE];
531:
532: public:
533:
534:     // Wiring list
535:
536:     TCpuObject *ResetList;           // List of latches.
537:     TDisplayObject *DisplayList;    // List of displayable things
538:
539:     // Data Registers
540:
541:     TRegister *A_Reg, *B_Reg, *Op_Reg, *Op_Mod_Reg;
542:
543:     // Address Registers
544:
545:     TAddressRegister *STAR;        // Storage Address Register
546:                                     // AKA MAR (Memory Address Register)
547:
548:     TAddressRegister *A_AR, *B_AR, *C_AR, *D_AR, *E_AR, *F_AR;
549:
550:     TAddressRegister *I_AR;        // Instruction Counter
551:
552:
553:     // Channels
554:
555:     T1410Channel *Channel[MAXCHANNEL]; // 2 I/O Channels.
556:
557:     TAChannel *AChannel;          // A Channel
558:     TAssemblyChannel *AssemblyChannel; // Assembly Channel
559:
560:     // Indicators
561:
562:     TDisplayIndicator *OffNormal; // OFF NORMAL Indicator
563:
564:     // Ring Counters
565:
566:     TRingCounter *IRing;          // Instruction decode ring
567:     TRingCounter *ARing;          // Address decode ring
568:     TRingCounter *ClockRing;      // Cycle Clock
569:     TRingCounter *ScanRing;       // Address Modification Mode
570:     TRingCounter *SubScanRing;    // Arithmetic Scan type
571:     TRingCounter *CycleRing;      // CPU Cycle type
572:
573:     // Latches with Indicators
574:
575:     TDisplayLatch *CarryIn;       // Carry latch
576:     TDisplayLatch *CarryOut;      // Adder has generated carry
577:     TDisplayLatch *AComplement;   // A channel complement
578:     TDisplayLatch *BComplement;   // B channel complement
579:     TDisplayLatch *CompareBGTA;  // B > A
580:     TDisplayLatch *CompareBEQA;  // B = A
581:     TDisplayLatch *CompareBLTA;  // B < A NOTE: On after C. Reset.
582:     TDisplayLatch *Overflow;      // Arithmetic Overflow
583:     TDisplayLatch *DivideOverflow; // Divide Overflow
584:     TDisplayLatch *ZeroBalance;   // Zero arithmetic result
585:
586:     // Check Latches
587:
588:     TDisplayLatch *AChannelCheck; // A Channel parity error
589:     TDisplayLatch *BChannelCheck; // B Channel parity error
590:     TDisplayLatch *AssemblyChannelCheck; // Assembly Channel parity error
591:     TDisplayLatch *AddressChannelCheck; // Address Channel parity error
592:     TDisplayLatch *AddressExitCheck; // Validity error at address reg.
593:     TDisplayLatch *ARegisterSetCheck; // A register failed to reset

```

```

594:     TDisplayLatch *BRegisterSetCheck;           // B register failed to reset
595:     TDisplayLatch *OpRegisterSetCheck;         // Op register failed to set
596:     TDisplayLatch *OpModifierSetCheck;         // Op modifier failed to set
597:     TDisplayLatch *ACharacterSelectCheck;      // Incorrect A channel gating
598:     TDisplayLatch *BCharacterSelectCheck;      // Incorrect B channel getting
599:
600:     TDisplayLatch *IOInterlockCheck;           // Program did not check I/O
601:     TDisplayLatch *AddressCheck;               // Program gave bad address
602:     TDisplayLatch *RBCInterlockCheck;          // Program did not check RBC
603:     TDisplayLatch *InstructionCheck;          // Program issued invalide op
604:
605: //  Switches
606:
607: enum Mode {                                     // Mode switch, values must match
608:     MODE_RUN = 0, MODE_DISPLAY = 1, MODE_ALTER = 2,
609:     MODE_CE = 3, MODE_IE = 4, MODE_ADDR = 5
610: } Mode;
611:
612: enum AddressEntry {
613:     ADDR_ENTRY_I = 0, ADDR_ENTRY_A = 1, ADDR_ENTRY_B = 2,
614:     ADDR_ENTRY_C = 3, ADDR_ENTRY_D = 4, ADDR_ENTRY_E = 5,
615:     ADDR_ENTRY_F = 6
616: } AddressEntry;
617:
618: //  The entry below is for the state of the 1415 Storage Scan SWITCH
619: //  (The storage scan modification mode is in the Ring ScanRing)
620:
621: enum StorageScan {
622:     SSCAN_OFF = 0, SSCAN_LOAD_1 = 1, SSCAN_LOAD_0 = 2,
623:     SSCAN_REGEN_0 = 3, SSCAN_REGEN_1 = 4
624: } StorageScan;
625:
626: enum CycleControl {
627:     CYCLE_OFF = 0, CYCLE_LOGIC = 1, CYCLE_STORAGE = 2
628: } CycleControl;
629:
630: enum CheckControl {
631:     CHECK_STOP = 0, CHECK_RESTART = 1, CHECK_RESET = 2
632: } CheckControl;
633:
634: bool DiskWrInhibit;
635: bool AsteriskInsert;
636: bool InhibitPrintOut;
637:
638: BCD BitSwitches;
639:
640: //  CPU  state latches
641:
642: bool StopLatch;                                // True to stop CPU
643: bool StopKeyLatch;                            // True if STOP button pressed
644: bool DisplayModeLatch;                      // True if we are displaying storage
645: bool ProcessRoutineLatch;        // True if we are in Run or IE mode
646: bool BranchLatch;                           // True if we are to branch
647: bool BranchTo1Latch;                        // True to branch to 1
648: bool LastInstructionReadout;    // True at end of Instruction fetch
649: bool IRingControl;                          // Set to start Instruction Fetch
650:
651: bool SignLatch;                             // Set if Minus sign at certain pts.
652: bool ZeroSuppressLatch;        // Set if supressing zeroes.
653: bool FloatingDollarLatch;    // Used in Edit
654: bool AsteriskFillLatch;      // Used in Edit
655: bool DecimalControlLatch;    // Used in Edit
656:
657: bool IOMoveModeLatch; // I/O Latches
658: bool IOLoadModeLatch; // to check for I/O Overlap
659:
    bool ImgRegLatch;                         // set for console info

```

```

660:     bool StorageWrapLatch;
661:
662:     int IOChannelSelect;           // One if if channel 2 op.
663:
664:     // Index latches are rolled up into an int. The BA zones from the
665:     // tens and hundreds positions are shifted and the 4 bits together
666:     // generate a value from 0 to 15.
667:
668:     int IndexLatches;
669:
670:     // Methods
671:
672:     T1410CPU();                  // Constructor
673:     void Display();              // Run thru the display list
674:     void Cycle();                // Used for common CPU Cycles
675:
676:     // The 1410 Adder accepts BCD inputs, a Carry Latch and complement
677:     // flags and returns a sum in BCD, possible setting Carry Out.
678:
679:     BCD Adder(BCD A,int Complement_A,BCD B,int Complement_B);
680:     BCD AdderResult;            // For the Assembly Channel
681:     void Comparator();
682: private:
683:
684:     // Indicator Routines
685:     // Normally, these will be accessed via a bool (_closure *func)()
686:
687:     bool IndicatorOffNormal();
688:     ✓ int AdderBinaryResult, AdderQuinaryResult;
689: public:
690:
691:     // Core operations (using STAR/MAR and B Data Register)
692:
693:     void Readout();             // Reads out one storage character
694:     void Store(BCD bcd);        // Store data
695:     void SetScan(char s);       // Sets Scan Modification value
696:
697:     long STARScan();           // Applies Scan CTRL modification to STAR,
698:                               // and returns the results - suitable to assign
699:
700:     long STARMod(int mod);     // Similar, but provides direct +1/0/-1 mod
701:
702:     void LoadCore(char *file); // Loads core from a file
703:     void DumpCore(char *file); // Dumps core to a file
704:
705: public:
706:     bool StorageWrap {
707:         // Instruction operations
708:
709:         unsigned short OpReadOutLines;
710:         unsigned short OpOperationalLines;
711:         unsigned short OpControlLines;
712:
713:         void DoStartClick();        // START pressed (moved from UI1410PWR)
714:         void InstructionDecodeStart(); // Starts instruction decode processing
715:         void InstructionDecode();    // Remainder of instruction decode
716:         void InstructionDecodeIARAdvance(); // Conditionally advance IAR
717:
718:         void InstructionIndexStart(); // Starts up indexing operation
719:         void InstructionIndex();     // Does the actual indexing
720:
721:         void InstructionExecuteInvalid(); // Default instruction execute routine
722:
723:         void InstructionArith();     // Add and Subtract
724:         void InstructionZeroArith(); // Zero and Add, Zero and Subtract
725:         void InstructionMultiply(); // Multiply

```

```
726:     void InstructionDivide();           // Divide
727:     void InstructionMove();             // Move
728:     void InstructionMoveSuppressZeros(); // Move and Suppress Zeros
729:     void InstructionEdit();            // Edit
730:
731: };
732:
733: extern T1410CPU *CPU;
734:
735: //-----
736: #endif
737:
```

```
1: //-----
2: #include <vc1.h>
3: #pragma hdrstop
4:
5: #include "UI1410CHANNEL.h"
6:
7: //-----
8: #pragma package(smart_init)
9:
10: #include <assert.h>
11: #include "UI1410CPUUT.H"
12: #include "UI1410INST.H"
13: #include "UI1415CE.H"
14:
15: // Implementation of I/O Channel Class
16:
17: // Constructor.  Initializes state
18:
19: T1410Channel::T1410Channel(
20:     TLabel *LampInterlock,
21:     TLabel *LampRBCInterlock,
22:     TLabel *LampRead,
23:     TLabel *LampWrite,
24:     TLabel *LampOverlap,
25:     TLabel *LampNotOverlap,
26:     TLabel *LampNotReady,
27:     TLabel *LampBusy,
28:     TLabel *LampDataCheck,
29:     TLabel *LampCondition,
30:     TLabel *LampWLRecord,
31:     TLabel *LampNoTransfer ) {
32:
33:     int i;
34:
35:     ChStatus = 0;
36:     TapeDensity = DENSITY_200_556;
37:     R1Status = R2Status = false;
38:
39:     ChStatusDisplay[IOLAMPNOTREADY] = LampNotReady;
40:     ChStatusDisplay[IOLAMPBUSY] = LampBusy;
41:     ChStatusDisplay[IOLAMPDATACHECK] = LampDataCheck;
42:     ChStatusDisplay[IOLAMPCONDITION] = LampCondition;
43:     ChStatusDisplay[IOLAMPNOTTRANSFER] = LampNoTransfer;
44:     ChStatusDisplay[IOLAMPWLRECORD] = LampWLRecord;
45:
46:     // Generally, the channel latches are *not* reset by Program Reset
47:
48:     ChInterlock = new TDisplayLatch(LampInterlock, false);
49:     ChRBCInterlock = new TDisplayLatch(LampRBCInterlock, false);
50:     ChRead = new TDisplayLatch(LampRead, false);
51:     ChWrite = new TDisplayLatch(LampWrite, false);
52:     ChOverlap = new TDisplayLatch(LampOverlap, false);
53:     ChNotOverlap = new TDisplayLatch(LampNotOverlap, false);
54:
55:     // Generally, the channel registers are *not* reset by Program Reset
56:
57:     ChOp = new TRegister(false);
58:     ChUnitType = new TRegister(false);
59:     ChUnitNumber = new TRegister(false);
60:     Chr1 = new TRegister(false);
61:     Chr2 = new TRegister(false);
62:
63:     // Clear the device table
64:
65:     for(i=0; i < 64; ++i) {
66:         Devices[i] = NULL;
```

```
67:     }
68:
69:     CurrentDevice = NULL;
70:     MoveMode = LoadMode = false;
71:     ExtEndofTransfer = IntEndofTransfer = false;
72:     CycleRequired = false;
73:     InputRequest = false;
74:     LastInputCycle = false;
75:     EndofRecord = false;
76: }
77:
78: // Channel Register methods
79:
80: BCD T1410Channel::SetR1(BCD b) {
81:     ChR1 -> Set(b);
82:     R1Status = true;
83:     return(b);
84: };
85:
86: BCD T1410Channel::SetR2(BCD b) {
87:     Chr2 -> Set(b);
88:     R2Status = true;
89:     return(b);
90: };
91:
92: // Move data from register 1 to register 2
93: // Clearing register 1 in the process.
94:
95: BCD T1410Channel::MoveR1R2() {
96:     Chr2 -> Set(ChR1 -> Get());
97:     R1Status = false;
98:     R2Status = true;
99:     return(Chr2 -> Get());
100: }
101:
102: // Method to add a new device to the channel's device table.
103: // Typically this is called from the T1410IODevice base class constructor
104:
105: void T1410Channel::AddIODevice(T1410IODevice *iodevice, int devicenumber) {
106:     assert(Devices[devicenumber] == NULL);
107:     Devices[devicenumber] = iodevice;
108: }
109:
110: // Channel is reset during ComputerReset
111:
112: void T1410Channel::OnComputerReset()
113: {
114:     Reset();
115: }
116:
117: void T1410Channel::Reset() {
118:     ChStatus = 0;
119:     R1Status = R2Status = false;
120:     MoveMode = LoadMode = false;
121:     ExtEndofTransfer = IntEndofTransfer = false;
122:     CycleRequired = false;
123:     InputRequest = false;
124:     LastInputCycle = false;
125:     EndofRecord = false;
126:     ChInterlock -> Reset();
127:     ChrBCInterlock -> Reset();
128:     ChRead -> Reset();
129:     ChWrite -> Reset();
130:     ChOverlap -> Reset();
131:     ChNotOverlap -> Reset();
132: }
```

```
133:
134: // Channel is not reset during Program Reset
135:
136: void T1410Channel::OnProgramReset()
137: {
138:     // Channel not affected by Program Reset
139: }
140:
141: // Display Routine.
142:
143: void T1410Channel::Display() {
144:
145:     int i;
146:
147:     ChStatusDisplay[IOLAMPNOTREADY] -> Enabled =
148:         ((ChStatus & IOCHNOTREADY) != 0);
149:     ChStatusDisplay[IOLAMPBUSY] -> Enabled =
150:         ((ChStatus & IOCHBUSY) != 0);
151:     ChStatusDisplay[IOLAMPDATACHECK] -> Enabled =
152:         ((ChStatus & IOCHDATACHECK) != 0);
153:     ChStatusDisplay[IOLAMPCONDITION] -> Enabled =
154:         ((ChStatus & IOCHCONDITION) != 0);
155:     ChStatusDisplay[IOLAMPWLRECORD] -> Enabled =
156:         ((ChStatus & IOCHWLRECORD) != 0);
157:     ChStatusDisplay[IOLAMPNOTTRANSFER] -> Enabled =
158:         ((ChStatus & IOCHNOTTRANSFER) != 0);
159:
160:     for(i=0; i <= 5; ++i) {
161:         ChStatusDisplay[i] -> Repaint();
162:     }
163:
164:     // Although in most instances the following would be redundant,
165:     // because these objects are also on the CPU display list, we include
166:     // them here in case we want to display a channel separately.
167:
168:     ChInterlock -> Display();
169:     ChrBCInterlock -> Display();
170:     ChRead -> Display();
171:     ChWrite -> Display();
172:     ChOverlap -> Display();
173:     ChNotOverlap -> Display();
174: }
175:
176: // Channel Lamp Test
177:
178: void T1410Channel::LampTest(bool b)
179: {
180:     int i;
181:
182:     // Note, we don't have to do anything to the TDisplayLatch objects in
183:     // the channel for lamp test. They will take care of themselves on a
184:     // lamp test.
185:
186:     if(!b) {
187:         for(i=0; i <= 5; ++i) {
188:             ChStatusDisplay[i] -> Enabled = true;
189:             ChStatusDisplay[i] -> Repaint();
190:         }
191:     }
192:     else {
193:         Display();
194:     }
195: }
196:
197: // Channel Output to Device
198: // Conditions on Entry: B data register (B_REG) contains character to output
```

```
199: // Channel Register 1 (E1 or F1) should be empty.
200:
201: void T1410Channel::DoOutput(TAddressRegister *addr) {
202:
203:     BCD tempbcd;
204:
205:     CPU -> CycleRing -> Set(this == CPU -> Channel[CHANNEL1] ? CYCLE_E : CYCLE_F);
206:
207:     // Read out the next character. Also, check for a storage wrap, which will
208:     // end the transfer after this character.
209:
210:     *(CPU -> STAR) = *addr;
211:     CPU -> Readout();
212:     CPU -> StorageWrapCheck(+1);
213:     tempbcd = CPU -> B_Reg -> Get();
214:
215:     // If we have a GMWM, then we are done transferring data from memory
216:     // Otherwise, Clear the WM from the input data if in Move mode, and
217:     // then (regardless of mode) transfer the data to Channel register R1
218:     // Also, if we actually have read a character out, check for storage
219:     // wrap, which also ends the transfer.
220:
221:     if(tempbcd.TestGMWM()) {
222:         IntEndofTransfer = true;
223:         EndofRecord = true;
224:     }
225:     else {
226:         if(MoveMode) {
227:             tempbcd.ClearWM();
228:             tempbcd.SetOddParity();
229:         }
230:         SetR1(tempbcd);
231:         if(CPU -> StorageWrapLatch) {
232:             IntEndofTransfer = true;
233:             EndofRecord = true;
234:         }
235:     }
236:
237:     // Now, if we have anything for the device, send it. We continue
238:     // doing this for up to two characters, in case R1 and R2 are both
239:     // full. We have to do it this way so that if, at the end, we set
240:     // ExtEndofTransfer, all of the data will have been sent.
241:
242:     while(GetR1Status() || GetR2Status()) {
243:         if(GetR1Status() & !GetR2Status()) {
244:             MoveR1R2();
245:         }
246:         CurrentDevice -> DoOutput();
247:         ResetR2();
248:     }
249:
250:     // Advance the appropriate address register to the next location.
251:     // (Storage wrap was detected earlier, as necessary!).
252:
253:     addr -> Set(CPU -> STARMOD(+1));
254:
255:     // Now that the data is all sent, if we are at IntEndofTransfer (either
256:     // From a GMWM or from storage wrap), also set ExtEndofTransfer so that
257:     // we quit. Note that the device may have already set ExtEndofTransfer!
258:
259:     if(IntEndofTransfer) {
260:         ExtEndofTransfer = true;
261:     }
262:
263: }
```

```
265: // Channel Input Processing (shared overlap/not overlap code)
266:
267: void T1410Channel::DoInput(TAddressRegister *addr) {
268:
269:     CycleRequired = false;                                // Reset cycle required
270:     InputRequest = false;                               // Reset co-routine flag
271:
272:     if(!IntEndofTransfer) {                            // Skip this if wrapped!
273:         *(CPU -> STAR) = *addr;                      // Copy memory address
274:         CPU -> Readout();                           // Get existing memory
275:         CPU -> AChannel -> Select();                // Gate A channel approp.
276:         if(this == CPU -> Channel[CHANNEL1] ?          TACchannel::A_Channel_E : TACchannel::A_Channel_F );
277:     }
278:
279:
280:     // If no data from device, the end is near...
281:
282:     if(!(GetR1Status() || GetR2Status())) {
283:         LastInputCycle = true;
284:     }
285:
286:     // If B GMWM, we are typically done storing data (unless this
287:     // is read to end of core, which ignores GMWM). Unless, of course,
288:     // we have already hit end of record.
289:
290:     // Note that we can only check the op mod if we are NOT overlapped
291:     // (Note that the "to end of core" mods are all not overlapped)
292:
293:     if((CPU -> Op_Mod_Reg -> Get() && CPU -> B_Reg -> Get().TestGMWM())
294:     {
295:         if(ChOverlap -> State() ||
296:             (CPU -> Op_Mod_Reg -> Get().To6Bit()) != OP_MOD_SYMBOL_DOLLAR) {
297:             EndofRecord = true;
298:         }
299:
300:         // If no more input, or we hit GMWM, or memory just wrapped,
301:         // set internal end of transfer, but keep on accepting characters
302:         // until External end of transfer.
303:
304:         if(LastInputCycle || EndofRecord !LastInputCycle & !EndofRecord) {
305:             IntEndofTransfer = true;
306:             if(ExtEndofTransfer) {
307:                 return;
308:             }
309:         }
310:
311:         // If we get here, we are either ending, or we are still storing data
312:
313:         if(!LastInputCycle && !IntEndofTransfer) {        // Still storing input?
314:
315:             // If parity is no good, set data check...
316:
317:             if(!CPU -> AChannel -> Select().CheckParity()) {
318:                 SetStatus(GetStatus() | IOCHDATACHECK);
319:
320:                 // If asterisk insert, store an asterisk!
321:
322:                 if(FI1415CE -> AsteriskInsert -> Checked) {
323:                     ChR2 -> Set(BCD_ASTERISK);
324:                     CPU -> AChannel -> Select();
325:                     if(this == CPU -> Channel[CHANNEL1] ?          TACchannel::A_Channel_E : TACchannel::A_Channel_F );
326:                 }
327:             } // End initial parity check
328:
329:
```

```
330:         // Store the data (perhaps with a parity error!
331:
332:         CPU -> Store(CPU -> AssemblyChannel -> Select(
333:             (MoveMode ?
334:                 TAssemblyChannel::AsmChannelWMB :
335:                 TAssemblyChannel::AsmChannelWMA),
336:                 TAssemblyChannel::AsmChannelZonesA,
337:                 false,
338:                 TAssemblyChannel::AsmChannelSignNone,
339:                 TAssemblyChannel::AsmChannelNumA) );
340:
341:         // If we have bad data, but not asterisk insert, STOP
342:         // One good way: Run FORTRAN w/o Asterisk Insert! ;)
343:
344:         if(!CPU -> AChannel -> Select().CheckParity()) {
345:             CPU -> AChannelCheck -> SetStop("Data Check, No Asterisk Insert!");
346:             return;
347:         }
348:
349:         ResetR2();                                // Reset Channel Data
350:         if( CPU -> StorageWrapCheck(+1)) {        // If storage wrap -- done
351:             IntEndofTransfer = true;
352:             } End of Record = true;
353:             else {
354:                 addr -> Set(CPU -> STARMOD(1));      // Bump address register
355:             }
356:
357:         } // End, !LastInputCycle
358:
359:         // In the real world, the device keeps reading data, and will
360:         // then set CycleRequired. (In fact, the console matrix will
361:         // call ChannelStrobe to actually do this when a key is pressed).
362:         // But most of our input devices in the emulator are co-routines,
363:         // so we have to call them back to give them a chance to strobe
364:         // the channel again. Also, in the real world, a device would
365:         // keep reading input data after the channel set Internal End of
366:         // Transfer -- we have to give the input co-routine a way to
367:         // finish reading it's record, even if we are no longer storing
368:         // it because we hit a GMWM.
369:
370:         CurrentDevice -> DoInput();
371:
372:     }
373:
374:     bool T1410Channel::ChannelStrobe(BCD ch) {
375:
376:         // If R1 has data already, move it to R2.
377:         // (In the emulator, that should probably never happen!)
378:
379:         if(GetR1Status()) {
380:             MoveR1R2();
381:         }
382:
383:         // Load R1, and copy to R2 if there is room in R2.
384:
385:         SetR1(ch);
386:         if(!GetR2Status()) {
387:             MoveR1R2();
388:         }
389:
390:         // If the channel has not already terminated the transfer,
391:         // ask to send the data to memory.
392:
393:         if(!IntEndofTransfer) {
394:             CycleRequired = true;
395:         }
```

```
396:
397:     InputRequest = true;                      // Force co-routine call
398:     return true;
399: }
400:
401: □
402:
403: // Class T1410IODevice implementation. This is an *abstract* base class,
404: // intended to be used to derive actual I/O devices
405:
406: T1410IODevice::T1410IODevice(int devicenumber, T1410Channel *Ch) {
407:     Channel = Ch;
408:     Ch -> AddIODevice(this,devicenumber);
409: }
410:
411: // And, finally, the 1410 IO Instruction Routines. We implement them
412: // here to keep the I/O stuff together!
413:
414: // Move and Load mode (M and L) Instructions.
415:
416: // Note: The channel selected by the CPU is known to be available, as the
417: // interlock test was passed during instruction readout at I3.
418: // IOChannelSelect indicates the selected channel.
419: // Channel -> ChUnitType has Device Type (e.g. 'T' for console)
420: // Channel -> ChUnitNumber has Unit Number
421:
422: void T1410CPU::InstructionIO() {
423:
424:     BCD opmod;
425:     T1410Channel *Ch = Channel[IOChannelSelect];
426:
427:     opmod = (Op_Mod_Reg -> Get()).To6Bit();
428:     assert(!(Ch -> ChInterlock -> State()));
429:
430:     // Reset the channel, then set Channel Interlock
431:
432:     Ch -> Reset();
433:     Ch -> ChInterlock -> Set();
434:
435:     // Set Move mode or Load mode, appropriately
436:
437:     if((Op_Reg -> Get()).To6Bit() == OP_IO_MOVE) {
438:         Ch -> MoveMode = true;
439:     }
440:     else {
441:         Ch -> LoadMode = true;
442:     }
443:
444:     // Start things out, depending on op modifier.
445:
446:     switch(opmod.ToInt()) {
447:
448:     case OP_MOD_SYMBOL_R:
449:     case OP_MOD_SYMBOL_DOLLAR:
450:         Ch -> ChRead -> Set();
451:         break;
452:
453:     case OP_MOD_SYMBOL_W:
454:     case OP_MOD_SYMBOL_X:
455:         Ch -> ChWrite -> Set();
456:         break;
457:
458:     default:
459:         InstructionCheck ->
460:             SetStop("Instruction Check: Invalid I/O d-character");
461:         return;
```

```
462:     } // End switch on op modifier
463:
464: // See if there is a device for this device number. If not,
465: // return not ready.
466:
467: if(Ch -> GetCurrentDevice() == NULL) {
468:     Ch -> SetStatus(IOCHNOTREADY);
469:     IRingControl = true;
470:     return;
471: }
472:
473: // Start up the I/O, do initial status check (Status Sample A)
474: // Just return if the status is not 0.
475:
476: Ch -> SetStatus(Ch -> GetCurrentDevice() -> Select());
477: if(Ch -> GetStatus() != 0) {
478:     IRingControl = true;
479:     return;
480: }
481:
482: // If OK so far, set Overlap/NotOverlap
483:
484: if(CPU -> IOOverlapSelect) {
485:     Ch -> ChOverlap -> Set();
486:     // TODO: Ch -> RequestOutput() or RequestInput (Write/Read)
487:     IRingControl = true;
488:     return;
489: }
490: else {
491:     Ch -> ChNotOverlap -> Set();
492: }
493:
494: // If we get here, we are not overlapped.
495:
496: if((opmod.ToInt() & OP_MOD_SYMBOL_W) == OP_MOD_SYMBOL_W) {
497:     CPU -> Display();
498:     while(!Ch -> ExtEndofTransfer) {
499:         Ch -> DoOutput(B_AR);
500:     }
501:     Ch -> ChNotOverlap -> Reset();
502:     Ch -> SetStatus(Ch -> GetCurrentDevice() -> StatusSample());
503:     IRingControl = true;
504:     return;
505: }
506: else if((opmod.ToInt() & OP_MOD_SYMBOL_R) == OP_MOD_SYMBOL_R) {
507:
508:     // Input processing continues so long as not External End from device
509:
510:     CPU -> Display();
511:     while(!Ch -> ExtEndofTransfer) {
512:
513:         // If no input, just wait here (not overlapped -- stuck here)
514:
515:         while(!(Ch -> CycleRequired || Ch -> InputRequest ||
516:                 Ch -> ExtEndofTransfer)) {
517:             Application -> ProcessMessages();
518:             // sleep(10);
519:             continue;
520:         }
521:
522:         Ch -> DoInput(B_AR);
523:
524:     } // End, not External End of Transfer
525:
526:     assert(Ch -> ExtEndofTransfer);
527:
```

```
528:     // If, at the end, things do not match up ==> Wrong Length Record
529:     // (Note that we do *not* test InputRequest here!)
530:
531:     Ch -> SetStatus(Ch -> GetCurrentDevice() -> StatusSample());
532:     if(Ch -> CycleRequired || Ch -> GetR2Status() || !Ch -> EndofRecord) {
533:         Ch -> SetStatus(Ch -> GetStatus() | IOCHWLRECORD);
534:     }
535:
536:     // And all done - continue with Instructions
537:
538:     Ch -> ChNotOverlap -> Reset();
539:     IRingControl = true;
540:     return;
541: }
542: }
543:
544:
```

```
1: //-----  
2: #ifndef UI1415IOH  
3: #define UI1415IOH  
4: //-----  
5: #include <vcl\Classes.hpp>  
6: #include <vcl\Controls.hpp>  
7: #include <vcl\StdCtrls.hpp>  
8: #include <vcl\Forms.hpp>  
9: //-----  
10:  
11: // 1415 Console User Interface  
12:  
13: #include "ubcd.h"  
14: #include "UI1410CPUUT.h"  
15:  
16: #include <vcl\ExtCtrls.hpp>  
17:  
18: #define CONSOLE_IDLE 1 // Console is idle  
19: #define CONSOLE_NORMAL 2 // Normal Read Mode input  
20: #define CONSOLE_LOAD 3 // Normal Load Mode input  
21: #define CONSOLE_ALTER 4 // Console is in alter mode loading memory  
22: #define CONSOLE_ADDR 5 // Console is in address set or display mode  
23: #define CONSOLE_DISPLAY 6 // Console is displaying register or mem  
24:  
25: #define CONSOLE_MATRIX_HOME 0 // Console matrix home position  
26:  
27: class TFI1415IO : public TForm  
28: {  
29:     __published: // IDE-managed Components  
30:         TMemo *I1415IO;  
31:         TLabel *KeyboardLock;  
32:         TButton *InqReq;  
33:         TButton *InqRlse;  
34:         TButton *InqCancel;  
35:         TButton *WordMark;  
36:         TButton *MarginRelease;  
37:         TTimer *KeyboardLockReset;  
38:         void __fastcall I1415IOKeyPress(TObject *Sender, char &Key);  
39:  
40:         void __fastcall I1415IOKeyDown(TObject *Sender, WORD &Key, TShiftState Shift);  
41:  
42:         void __fastcall WordMarkClick(TObject *Sender);  
43:         void __fastcall MarginReleaseClick(TObject *Sender);  
44:  
45:         void __fastcall KeyboardLockResetTimer(TObject *Sender);  
46:  
47:     private: // User declarations  
48:  
49:         int state;  
50:         int matrix;  
51:  
52:         bool WMCTrlLatch;  
53:         bool DisplayWMCTrlLatch;  
54:  
55:         bool AlterFullLineLatch;  
56:         bool AlterWMConditionLatch;  
57:  
58:         TAddressRegister *Console_AR;  
59:  
60:         bool DoWordMark();  
61:         void LockKeyboard();  
62:         void UnlockKeyboard();  
63:  
64:     public: // User declarations  
65:         __fastcall TFI1415IO(TComponent* Owner);  
66:
```

```
67:     void SendBCDTo1415(BCD bcd);
68:     bool SetState(int s);
69:     void NextLine();
70:
71:     inline void SetMatrix(int i) { matrix = i; }
72:     void SetMatrix(int x, int y) { matrix = 6*(x-1) + y; }
73:     inline int GetMatrix() { return matrix; }
74:     inline int GetMatrixX() { return (matrix-1)/6 + 1; }
75:     inline int GetMatrixY() { return (matrix-1)%6 + 1; }
76:     void StepMatrix() { ++matrix; }
77:     void ResetMatrix() { matrix = CONSOLE_MATRIX_HOME; }
78:     void DoMatrix();
79:
80:     void StopPrintOut(char c);
81:     void DoAddressEntry();
82:     void DoDisplay(int phase);
83:     void DoAlter();
84:
85: };
86: //-----
87: extern TFI1415IO *FI1415IO;
88: //-----
89:
90: //  Keyboard representations of some unusual BCD characters
91:
92: #define KBD_RADICAL          022
93: #define KBD_RECORD_MARK      '|'
94: #define KBD_ALT_BLANK        002
95: #define KBD_WORD_SEPARATOR   '^'
96: #define KBD_SEGMENT_MARK    023
97: #define KBD_DELTA             004
98: #define KBD_GROUP_MARK       007
99: #define KBD_WORD_MARK        0x1b
100:
101: #endif
```

```
1: // This Unit defines the behavior of the 1415 console typewriter and
2: // associated things
3:
4: -----
5: #include <vcl\vcl.h>
6: #pragma hdrstop
7:
8: #include "UI1415IO.h"
9: #include "ubcd.h"
10: #include "UI1410DEBUG.h"
11: #include "UI1410CPUT.h"
12:
13: -----
14: #pragma resource "* .dfm"
15: TFI1415IO *FI1415IO;
16: -----
17: __fastcall TFI1415IO::TFI1415IO(TComponent* Owner)
18:   : TForm(Owner)
19: {
20:   Height = 333;           { Top=0;
21:   Width = 599;            { Left=0;
22:
23:   state = CONSOLE_IDLE;
24:   Console_AR = 0;
25:   WMCtrlLatch = false;
26:   DisplayWMCtrlLatch = false;
27:   AlterFullLineLatch = false;
28:   AlterWMConditionLatch = false;
29: }
30: -----
31: void __fastcall TFI1415IO::I1415IOKeyPress(TObject *Sender, char &Key)
32: {
33:   BCD bcd_key;
34:   static last_key_was_wm;
35:
36:   switch(Key) {
37:   case KBD_RADICAL:
38:     bcd_key = BCD::BCDConvert(ASCII_RADICAL);
39:     break;
40:   case KBD_RECORD_MARK:
41:     bcd_key = BCD::BCDConvert(ASCII_RECORD_MARK);
42:     break;
43:   case KBD_ALT_BLANK:
44:     bcd_key = BCD::BCDConvert(ASCII_ALT_BLANK);
45:     break;
46:   case KBD_WORD_SEPARATOR:
47:     bcd_key = BCD::BCDConvert(ASCII_WORD_SEPARATOR);
48:     break;
49:   case KBD_SEGMENT_MARK:
50:     bcd_key = BCD::BCDConvert(ASCII_SEGMENT_MARK);
51:     break;
52:   case KBD_DELTA:
53:     bcd_key = BCD::BCDConvert(ASCII_DELTA);
54:     break;
55:   case KBD_GROUP_MARK:
56:     bcd_key = BCD::BCDConvert(ASCII_GROUP_MARK);
57:     break;
58:   case KBD_WORD_MARK: // Escape == Wordmark Key
59:     DoWordMark();
60:     last_key_was_wm = true;
61:     return;
62:   case 'b':
63:     bcd_key = BCD::BCDConvert('B');
64:     break;
65:   default:
66:     if(BCD::BCDCheck(Key) < 0) {           // Return if unmapped key.
```

```
67:             LockKeyboard();
68:             return;
69:         }
70:         bcd_key = BCD::BCDConvert(Key);
71:     }
72:
73:     if(last_key_was_wm) {
74:         last_key_was_wm = false;
75:         bcd_key.SetWM();
76:     }
77:
78:     // Decide what to do with key, depending on console state.
79:     // Note that the really special keys (Wordmark, above, and the
80:     // console inquiry buttons), are checked elsewhere. This test
81:     // is for "normal" BCD characters only.
82:
83:     switch(state) {
84:
85:     case CONSOLE_IDLE:
86:         LockKeyboard();                                // Only INQ Request allowed
87:         return;
88:
89:     case CONSOLE_NORMAL:                           // All keys allowed
90:     case CONSOLE_LOAD:
91:         break;
92:
93:     case CONSOLE_ALTER:
94:         UnlockKeyboard();
95:         SendBCDTo1415(bcd_key);
96:
97:         // Ensure that the incoming character is odd parity
98:
99:         bcd_key.SetOddParity();
100:        DEBUG("CONSOLE ALTER: Key is %d", bcd_key.ToInt())
101:
102:        // During an ALTER operation, the E Channel (channel 1) gets
103:        // the character on its way to the address register (via the
104:        // A channel and Assembly channel)
105:
106:        CPU -> Channel[CHANNEL1] -> ChR1 -> Set(bcd_key);
107:        *(CPU -> Channel[CHANNEL1] -> ChR2) =
108:            *(CPU -> Channel[CHANNEL1] -> ChR1);
109:
110:        DoMatrix();
111:
112:        break;
113:
114:    case CONSOLE_ADDR:                            // Only digits allowed
115:        if(isdigit(Key)) {
116:            UnlockKeyboard();
117:            SendBCDTo1415(bcd_key);
118:
119:            // Ensure that the incoming character has the correct (odd) parity
120:
121:            bcd_key.SetOddParity();
122:
123:            DEBUG("BCD Key set to %d", bcd_key.ToInt())
124:
125:            // During an ADDR SET operation, the E Channel (channel 1) gets
126:            // the character on its way to the address register (via the
127:            // A channel and Assembly channel)
128:
129:            CPU -> Channel[CHANNEL1] -> ChR1 -> Set(bcd_key);
130:            *(CPU -> Channel[CHANNEL1] -> ChR2) =
131:                *(CPU -> Channel[CHANNEL1] -> ChR1);
132:            DoMatrix();
```

```
133:         }
134:     else {
135:         LockKeyboard();
136:     }
137:     break;
138: }
139: }
140: //-----
141: void __fastcall TFI1415IO::I1415IOKeyDown(TObject *Sender, WORD &Key,
142:                                         TShiftState Shift)
143: {
144:     // Special key handling: Treat Page Down as a Margin Release.
145:     if(Key == VK_NEXT) {
146:         NextLine();
147:     }
148: }
149: }
150: }
151: //-----
152: 
153: // Method to set the state of the console
154:
155: bool TFI1415IO::SetState(int s)
156: {
157:     switch(s) {
158:     case CONSOLE_IDLE:
159:         InqReq -> Enabled = true;
160:         InqRlse -> Enabled = false;
161:         InqCancel -> Enabled = false;
162:         WordMark -> Enabled = false;
163:         CPU -> DisplayModeLatch = false;
164:         break;
165:     case CONSOLE_NORMAL:
166:         InqReq -> Enabled = true;
167:         InqRlse -> Enabled = true;
168:         InqCancel -> Enabled = true;
169:         WordMark -> Enabled = false;
170:         break;
171:     case CONSOLE_LOAD:
172:         InqReq -> Enabled = true;
173:         InqRlse -> Enabled = true;
174:         InqCancel -> Enabled = true;
175:         WordMark -> Enabled = true;
176:         break;
177:     case CONSOLE_ALTER:
178:         InqReq -> Enabled = false;
179:         InqRlse -> Enabled = false;
180:         InqCancel -> Enabled = false;
181:         WordMark -> Enabled = true;
182:         break;
183:     case CONSOLE_ADDR:
184:         InqReq -> Enabled = false;
185:         InqRlse -> Enabled = false;
186:         InqCancel -> Enabled = false;
187:         WordMark -> Enabled = false;
188:         switch(CPU -> AddressEntry) {
189:             case CPU -> ADDR_ENTRY_I:
190:                 Console_AR = CPU -> I_AR;
191:                 break;
192:             case CPU -> ADDR_ENTRY_A:
193:                 Console_AR = CPU -> A_AR;
194:                 break;
195:             case CPU -> ADDR_ENTRY_B:
196:                 Console_AR = CPU -> B_AR;
197:                 break;
198:             case CPU -> ADDR_ENTRY_C:
```

```
199:         Console_AR = CPU -> C_AR;
200:         break;
201:     case CPU -> ADDR_ENTRY_D:
202:         Console_AR = CPU -> D_AR;
203:         break;
204:     case CPU -> ADDR_ENTRY_E:
205:         Console_AR = CPU -> E_AR;
206:         break;
207:     case CPU -> ADDR_ENTRY_F:
208:         Console_AR = CPU -> F_AR;
209:         break;
210:     default:
211:         DEBUG("1415IO: Invalid CPU Address Entry case: %d",CPU->AddressEntry);
212:         Console_AR = NULL;
213:     }
214:     if(CPU -> DisplayModeLatch == true) {
215:         Console_AR = CPU -> C_AR;
216:     }
217:     break;
218: case CONSOLE_DISPLAY:
219:     InqReq -> Enabled = false;
220:     InqRlse -> Enabled = false;
221:     InqCancel -> Enabled = false;
222:     WordMark -> Enabled = false;
223:     break;
224: default:
225:     return(false);
226: }
227: state = s;
228: return(true);
229: }
230:
231:
232: /**
233: // Utility routine to go to next line on console
234: //
235:
236: void TFI1415IO::NextLine()
237: {
238:     I1415IO -> Lines -> Add("");    // Placeholder for wordmarks
239:     I1415IO -> Lines -> Add("");
240:
241:     CPU -> Display();
242: }
243:
244: /**
245: // Utility routines to lock/unlock keyboard - display/clear light.
246: //
247:
248: void TFI1415IO::LockKeyboard()
249: {
250:     KeyboardLock -> Enabled = true;
251:     KeyboardLockReset -> Enabled = true;
252: }
253:
254: void TFI1415IO::UnlockKeyboard()
255: {
256:     KeyboardLock -> Enabled = false;
257:     KeyboardLockReset -> Enabled = false;
258: }
259:
260: /**
261: // Utility routine to handle wordmark key
262: // Returns true if wordmark was valid
263: //
264:
```

```
265: bool TFI1415IO::DoWordMark()
266: {
267:     int last;
268:
269:     if(state != CONSOLE_LOAD && state != CONSOLE_ALTER) {
270:         LockKeyboard();
271:         return(false);
272:     }
273:
274:     last = I1415IO -> Lines -> Count - 1;
275:     if(last <= 0) {
276:         NextLine();
277:         last = I1415IO -> Lines -> Count -1;
278:     }
279:
280:     if(I1415IO -> Lines -> Strings[last-1].Length() <=
281:         I1415IO -> Lines -> Strings[last].Length() ) {
282:         I1415IO -> Lines -> Strings[last-1] =
283:             I1415IO -> Lines -> Strings[last-1] + "v";
284:         UnlockKeyboard();
285:         return(true);
286:     }
287:     else {
288:         LockKeyboard();
289:         return(false);
290:     }
291: }
292:
293: /**
294: // Utility routine to send data to console from
295: //
296:
297: void TFI1415IO::SendBCDTo1415(BCD bcd)
298: {
299:     int last;
300:     char c;
301:
302:     // Advance to next line if this is the very first line, in
303:     // order to make room for first line of wordmarks
304:
305:     last = I1415IO -> Lines -> Count - 1;
306:     if(last <= 0) {
307:         NextLine();
308:         last = I1415IO -> Lines -> Count -1;
309:     }
310:
311:     // Advance wordmark line to current position if necessary
312:
313:     if(I1415IO -> Lines -> Strings[last-1].Length() <=
314:         I1415IO -> Lines -> Strings[last].Length()) {
315:         I1415IO -> Lines -> Strings[last-1] =
316:             I1415IO -> Lines -> Strings[last-1] +
317:             (bcd.TestWM() ? "v" : " ");
318:     }
319:
320:     // If in display mode (registers or memory), print space as 'b'
321:     // Otherwise, just put the character into the line. (Using ToAscii
322:     // helps us to be invariant about WordMarks and Parity in this
323:     // situation)
324:
325:     c = (state == CONSOLE_DISPLAY && bcd.ToAscii() == ' ') ?
326:         'b' : bcd.ToAscii();
327:
328:     I1415IO -> Lines -> Strings[last] =
329:         I1415IO -> Lines -> Strings[last] + c;
330:
```

```
331: // Advance to next line if this line is full
332: // If we are doing a DISPLAY, set up the alter latches for later ALTER
333: // If we are doing an ALTER, this will terminate it in matrix 33.
334:
335: if(I1415IO -> Lines -> Strings[last].Length() > 79) {
336:     if(state == CONSOLE_DISPLAY) {
337:         AlterFullLineLatch = true;
338:         AlterWMConditionLatch = false;
339:     }
340:     if(state == CONSOLE_ALTER) {
341:         AlterFullLineLatch = false;
342:         AlterWMConditionLatch = false;
343:     }
344: }
345:
346: // Tell the form object it has been modified.
347:
348: I1415IO -> Modified = true;
349:
350: // Do a full display
351:
352: // Then again, ..... CPU -> Display();
353:
354: // Give Windoze a chance to breath
355:
356: Application -> ProcessMessages();
357:
358: }
359:
360: void TFI1415IO::DoMatrix()
361: {
362:     int savestate;
363:     BCD bcd_char;
364:
365:     if(matrix < 0 || matrix > 42) {
366:         DEBUG("Invalid Console Matrix Position: %d",matrix);
367:         return;
368:     }
369:
370: //     DEBUG("Doing Console Matrix Position %d",matrix);
371:
372: //     Console Matrix Processing
373:
374: //     Positions 0 (Home) thru 36 are used for display/output purposes
375:
376:     switch(matrix) {
377:
378:     case 0:
379:         // Home position: do nothing
380:         SetState(CONSOLE_IDLE);
381:         break;
382:
383:     case 1:
384:     case 2:
385:     case 3:
386:     case 4:
387:     case 5:
388:         SendBCDTo1415(CPU -> I_AR -> GateBCD(matrix));
389:         break;
390:
391:     case 6:
392:     case 12:
393:     case 18:
394:     case 21:
395:     case 25:
396:     case 31:
```

```
397:     case 36:
398:         savestate = state;
399:         state = CONSOLE_NORMAL;           // So space prints as one !!
400:         SendBCDTo1415(BCD::BCDConvert(' '));
401:         state = savestate;
402:         break;
403:
404:     case 7:
405:     case 8:
406:     case 9:
407:     case 10:
408:     case 11:
409:         SendBCDTo1415(CPU -> A_AR -> GateBCD(matrix-6));
410:         break;
411:
412:     case 13:
413:     case 14:
414:     case 15:
415:     case 16:
416:     case 17:
417:         SendBCDTo1415(CPU -> B_AR -> GateBCD(matrix-12));
418:         break;
419:
420:     case 19:
421:         SendBCDTo1415(CPU -> Op_Reg -> Get());
422:         break;
423:
424:     case 20:
425:         SendBCDTo1415(CPU -> Op_Mod_Reg -> Get());
426:         break;
427:
428:     case 22:
429:         SendBCDTo1415(CPU -> A_Reg -> Get());
430:         break;
431:
432:     case 23:
433:         SendBCDTo1415(CPU -> B_Reg -> Get());
434:         break;
435:
436:     case 24:
437:         SendBCDTo1415(CPU -> AssemblyChannel-> Select());
438:         break;
439:
440:     case 26:
441:         SendBCDTo1415(CPU -> Channel[CHANNEL1] -> ChUnitType -> Get());
442:         break;
443:
444:     case 27:
445:         SendBCDTo1415(CPU -> Channel[CHANNEL1] -> ChUnitNumber -> Get());
446:         break;
447:
448:     case 28:
449: #if MAXCHANNEL > 1
450:         SendBCDTo1415(CPU -> Channel[CHANNEL2] -> ChUnitType -> Get());
451: #endif
452:         break;
453:
454:     case 29:
455: #if MAXCHANNEL > 1
456:         SendBCDTo1415(CPU -> Channel[CHANNEL2] -> ChUnitNumber -> Get());
457: #endif
458:         break;
459:
460:     case 30:
461:         if(CPU -> DisplayModeLatch) {
462:             SendBCDTo1415(BCD::BCDConvert('D'));
```

```

463:     }
464:     else if(state == CONSOLE_ALTER) {
465:         SendBCDTo1415(BCD::BCDConvert('A'));
466:     }
467:     break;
468:
469: case 32:
470:     if(CPU -> DisplayModeLatch) {
471:         SendBCDTo1415(CPU -> B_Reg -> Get());
472:         WMCtrlLatch = CPU -> B_Reg -> Get().TestWM();
473:         StepMatrix();
474:     }
475:     else if(state == CONSOLE_ALTER) {
476:         CPU -> SetScan(SCAN_2);
477:         CPU -> CycleRing -> Set(CYCLE_D);
478:         *(CPU -> STAR) = *(CPU -> C_AR);
479:         CPU -> Store(CPU -> AssemblyChannel -> GateAChannelToAssembly(
480:             CPU -> AChannel -> Select(CPU -> AChannel -> A_Channel_E)));
481:         CPU -> D_AR -> Set(CPU -> STARScan());
482:         CPU -> CycleRing -> Set(CYCLE_D);
483:         CPU -> SetScan(SCAN_N);
484:         *(CPU -> STAR) = *(CPU -> D_AR);
485:         CPU -> Readout();
486:         if(CPU -> B_Reg -> Get().TestWM()) {
487:             AlterWMConditionLatch = false;
488:         }
489:         StepMatrix();
490:         if(!(AlterWMConditionLatch || AlterFullLineLatch)) {
491:             StepMatrix(); // Right to 34
492:             DoMatrix();
493:             ResetMatrix();
494:             SetState(CONSOLE_IDLE);
495:         }
496:     }
497:     break;
498:
499: case 33:
500:     if(CPU -> DisplayModeLatch) {
501:         SendBCDTo1415(CPU -> B_Reg -> Get());
502:         WMCtrlLatch = CPU -> B_Reg -> Get().TestWM();
503:         if(WMCtrlLatch && !AlterFullLineLatch) {
504:             AlterWMConditionLatch = true;
505:         }
506:         DisplayWMCtrlLatch = WMCtrlLatch;
507:     }
508:     else if(state == CONSOLE_ALTER) {
509:         CPU -> SetScan(SCAN_2);
510:         CPU -> CycleRing -> Set(CYCLE_D);
511:         *(CPU -> STAR) = *(CPU -> D_AR);
512:         CPU -> Store(CPU -> AssemblyChannel -> GateAChannelToAssembly(
513:             CPU -> AChannel -> Select(CPU -> AChannel -> A_Channel_E)));
514:         CPU -> D_AR -> Set(CPU -> STARScan());
515:         CPU -> CycleRing -> Set(CYCLE_D);
516:         CPU -> SetScan(SCAN_N);
517:         *(CPU -> STAR) = *(CPU -> D_AR);
518:         CPU -> Readout();
519:         if(CPU -> B_Reg -> Get().TestWM()) {
520:             AlterWMConditionLatch = false;
521:         }
522:         if(!(AlterWMConditionLatch || AlterFullLineLatch)) {
523:             StepMatrix(); // Right to 34
524:             DoMatrix();
525:             ResetMatrix();
526:             SetState(CONSOLE_IDLE);
527:         }
528:     }

```

else if(ConsoleIOInProgress){  
 if(CPU->Channel[Channel1] ->  
 ChRead->Stat()) {  
 SendBWTo1415(AddressOfData[1]);  
 }  
 R

```
529:         break;
530:
531:     case 34:
532:         NextLine();
533:         break;
534:
535: //  Positions 37 thru 42 are to accept an address for address set.
536: //  For 41, do the last digit, then fall thru to processing for 42.
537:
538:     case 37:
539:     case 38:
540:     case 39:
541:     case 40:
542:     case 41:
543:         bcd_char = CPU -> AssemblyChannel -> GateAChannelToAssembly(
544:             CPU -> AChannel -> Select(CPU -> AChannel -> A_Channel_E));
545:         DEBUG("Value to set into register from console is %d", bcd_char.ToInt());
546:         Console_AR -> Set(TWOOF5(bcd_char), matrix-36);
547:         StepMatrix();
548:         if(matrix != 42) {
549:             break;
550:         }
551:
552:         // Fall thru to next entry for last digit!!
553:
554: // At the end of an address set, go to next line, and reset console
555:
556:     case 42:
557:         NextLine();
558:         if(CPU -> DisplayModeLatch) {
559:             SetState(CONSOLE_DISPLAY);
560:             DoDisplay(2);
561:         }
562:         else {
563:             ResetMatrix();
564:             SetState(CONSOLE_IDLE);
565:         }
566:         break;
567:
568:     default:
569:         break;
570:     }
571:
572: //    CPU -> Display();
573: }
574:
575: // Handle a console stop print-out operation
576:
577: void TFI1415IO::StopPrintOut(char c)
578: {
579:     SetState(CONSOLE_DISPLAY);
580:     NextLine();
581:     SetMatrix(35);
582:     SendBCDTo1415(BCD::BCDConvert(c));
583:     StepMatrix();
584:     DoMatrix();
585:     SetMatrix(1);
586:     while(GetMatrixX() < 5) {
587:         DoMatrix();
588:         StepMatrix();
589:     }
590:
591:     while(GetMatrixY() < 6) {
592:         DoMatrix();
593:         StepMatrix();
594:     }
```

```

595:     }
596:     SetMatrix(34);
597:     DoMatrix();
598:     SetMatrix(CONSOLE_MATRIX_HOME);
599:     DoMatrix();
600:     CPU -> Display();
601: }
602:
603: // Initiate a Console Address Set operation
604:
605: void TFI1415IO::DoAddressEntry()
606: {
607:     SetState(CONSOLE_ADDR);
608:     Console_AR -> Reset();
609:     SetMatrix(35);
610:     if(CPU -> DisplayModeLatch) {
611:         SendBCDTo1415(BCD::BCDConvert('D'));
612:     }
613:     else {
614:         SendBCDTo1415(BCD::BCDConvert((Console_AR == CPU -> I_AR) ? 'B' : '#'));
615:     }
616:     StepMatrix();
617:     DoMatrix();
618:     SetMatrix(37);
619:
620:     // The console matrix code will automatically handle the rest, so
621:     // just return!
622:
623: }
624:
625: // Initiate a Memory Display operation
626:
627: void TFI1415IO::DoDisplay(int phase)
628: {
629:     // Phase 1 takes us thru the address entry. When the address entry
630:     // is done, the code in the matrix notices we are doing a display,
631:     // and fires up Phase 2 (like co-routines).
632:
633:     // We also come here if START is pressed to continue a display, in
634:     // which case the DisplayWMCtrlLatch will be set.
635:
636:     if(phase == 1) {
637:         if(DisplayWMCtrlLatch) {
638:             DisplayWMCtrlLatch = false;
639:             phase = 3;
640:         }
641:     else {
642:         CPU -> StopKeyLatch = false;
643:         CPU -> IRing -> Reset();
644:         WMCtrlLatch = DisplayWMCtrlLatch =
645:             AlterWMConditionLatch = AlterFullLineLatch = false;
646:
647:             // The DisplayModeLatch causes No A Ch, B Ch, Address Wrap Checks
648:             // It also forces the Address Entry routine to print 'D', and put
649:             // the resultant address into CAR
650:
651:             CPU -> DisplayModeLatch = true;
652:
653:             DoAddressEntry();
654:             return;
655:     }
656: }
657:
658: // Phase 2 handles the beginning of the actual display part thru matrix
659: // position 31.
660:
```

(5) Window State = wsNormal;

SetFocus();

BringToFront();

SA

```
661:     if(phase == 2) {
662:         matrix = 30;
663:         DoMatrix();                                // Prints "D"
664:         StepMatrix();
665:         DoMatrix();                                // Prints a space
666:         CPU -> SetScan(SCAN_2);                  // Read out first character
667:         CPU -> CycleRing -> Set(CYCLE_D);
668:         *(CPU -> STAR) = *(CPU -> C_AR);
669:         CPU -> Readout();
670:         StepMatrix();                            // Step to matrix position 32
671:         phase = 3;
672:     }
673:
674:     // Phase 3 handles normal circumstances for matrix position 33.  The
675:     // display is stopped by the STOP key (or moving the MODE switch)
676:     // (which goes to phase 4), or by hitting a wordmark, or by wrapping
677:     // storage.  (The POO and the CE Instructional materials disagree on
678:     // this last point - the POO says it keeps going, the Instructional
679:     // materials say the display stops)
680:
681:     if(phase == 3) {
682:         while(true) {
683:             if(DisplayWMCtrlLatch) {
684:                 break;
685:             }
686:             DoMatrix();
687:             CPU -> D_AR -> Set(CPU -> STARSscan());
688:             CPU -> SetScan(SCAN_2);
689:             CPU -> CycleRing -> Set(CYCLE_D);
690:             *(CPU -> STAR) = *(CPU -> D_AR);
691:
692:             // Check for any of: STOP Key/MODE Change, Address Wrap.
693:
694:             if(CPU -> StopKeyLatch) {
695:                 phase = 4;
696:                 break;
697:             }
698:             else {
699:                 CPU -> Readout();
700:                 if(CPU -> StorageWrapLatch) {
701:                     phase = 4;
702:                     CPU -> StopKeyLatch = true;
703:                     break;
704:                 }
705:             }
706:         }
707:     }
708:
709:     if(phase == 4 && (state == CONSOLE_DISPLAY || state == CONSOLE_ALTER)) {
710:         DisplayWMCtrlLatch = false;
711:         if(state == CONSOLE_ALTER) {
712:             AlterWMConditionLatch = false;
713:         }
714:         StepMatrix();                                // To Position 34.
715:         DoMatrix(); ↴
716:         ResetMatrix();
717:         SetState(CONSOLE_IDLE);
718:     }
719: }
720:
721: // Initiate a memory alter operation
722: // A DISPLAY must have happened first (we check the latches), so that
723: // CAR has the starting address, and either AlterFullLineLatch or
724: // AlterWMConditionLatch is set
725:
726: void TFI1415IO::DoAlter()
```

```
727: {  
728:     if(!(AlterWMConditionLatch || AlterFullLineLatch)) {  
729:         DEBUG("Cannot start ALTER without doing DISPLAY FIRST.",0)  
730:         return;  
731:     }  
732:     SetState(CONSOLE_ALTER); // S  
733:     matrix = 30;           // Print "A"  
734:     DoMatrix();  
735:     StepMatrix();        // Print a space  
736:     DoMatrix();  
737:     StepMatrix();        // Advance matrix to 32  
738:     DoMatrix();  
739:     // Now we wait for characters...  
740:     StepMatrix();  
741:     // NOTE: STOP KEY processing for Alter is handled in DoDisplay(4) !!  
742:     DoMatrix();  
743:     // Now we wait for characters...  
744:     StepMatrix();  
745:     DoMatrix();  
746: }  
747:  
748: void __fastcall TFI1415IO::WordMarkClick(TObject *Sender)  
749: {  
750:     DoWordMark();  
751:     FocusControl(I1415IO);  
752: }  
753: //-----  
754: void __fastcall TFI1415IO::MarginReleaseClick(TObject *Sender)  
755: {  
756:     NextLine();  
757:     FocusControl(I1415IO);  
758:     if(state == CONSOLE_DISPLAY) {  
759:         AlterFullLineLatch = true;  
760:         AlterWMConditionLatch = false;  
761:     }  
762: }  
763: //-----  
764: void __fastcall TFI1415IO::KeyboardLockResetTimer(TObject *Sender)  
765: {  
766:     UnlockKeyboard();  
767: }  
768: //-----
```

```
1: //-----
2: #include <vc1.h>
3: #pragma hdrstop
4:
5: #include "UI1410CHANNEL.h"
6:
7: //-----
8: #pragma package(smart_init)
9:
10: #include <assert.h>
11: #include "UI1410CPUH.H"
12: #include "UI1410INST.H"
13: #include "UI1415CE.H"
14:
15: // Implementation of I/O Channel Class
16:
17: // Constructor.  Initializes state
18:
19: T1410Channel::T1410Channel(
20:     TLabel *LampInterlock,
21:     TLabel *LampRBCInterlock,
22:     TLabel *LampRead,
23:     TLabel *LampWrite,
24:     TLabel *LampOverlap,
25:     TLabel *LampNotOverlap,
26:     TLabel *LampNotReady,
27:     TLabel *LampBusy,
28:     TLabel *LampDataCheck,
29:     TLabel *LampCondition,
30:     TLabel *LampWLRecord,
31:     TLabel *LampNoTransfer ) {
32:
33:     int i;
34:
35:     ChStatus = 0;
36:     TapeDensity = DENSITY_200_556;
37:     R1Status = R2Status = false;
38:
39:     ChStatusDisplay[IOLAMPNOTREADY] = LampNotReady;
40:     ChStatusDisplay[IOLAMPBUSY] = LampBusy;
41:     ChStatusDisplay[IOLAMPDATACHECK] = LampDataCheck;
42:     ChStatusDisplay[IOLAMPCONDITION] = LampCondition;
43:     ChStatusDisplay[IOLAMPNOTTRANSFER] = LampNoTransfer;
44:     ChStatusDisplay[IOLAMPWLRECORD] = LampWLRecord;
45:
46:     // Generally, the channel latches are *not* reset by Program Reset
47:
48:     ChInterlock = new TDisplayLatch(LampInterlock, false);
49:     ChRBCInterlock = new TDisplayLatch(LampRBCInterlock, false);
50:     ChRead = new TDisplayLatch(LampRead, false);
51:     ChWrite = new TDisplayLatch(LampWrite, false);
52:     ChOverlap = new TDisplayLatch(LampOverlap, false);
53:     ChNotOverlap = new TDisplayLatch(LampNotOverlap, false);
54:
55:     // Generally, the channel registers are *not* reset by Program Reset
56:
57:     ChOp = new TRegister(false);
58:     ChUnitType = new TRegister(false);
59:     ChUnitNumber = new TRegister(false);
60:     Chr1 = new TRegister(false);
61:     Chr2 = new TRegister(false);
62:
63:     // Clear the device table
64:
65:     for(i=0; i < 64; ++i) {
66:         Devices[i] = NULL;
```

```
67:     }
68:
69:     CurrentDevice = NULL;
70:     MoveMode = LoadMode = false;
71:     ExtEndofTransfer = IntEndofTransfer = false;
72:     CycleRequired = false;
73:     InputRequest = false;
74:     LastInputCycle = false;
75:     EndofRecord = false;
76: }
77:
78: // Channel Register methods
79:
80: BCD T1410Channel::SetR1(BCD b) {
81:     ChR1 -> Set(b);
82:     R1Status = true;
83:     return(b);
84: }
85:
86: BCD T1410Channel::SetR2(BCD b) {
87:     ChR2 -> Set(b);
88:     R2Status = true;
89:     return(b);
90: }
91:
92: // Move data from register 1 to register 2
93: // Clearing register 1 in the process.
94:
95: BCD T1410Channel::MoveR1R2() {
96:     ChR2 -> Set(ChR1 -> Get());
97:     R1Status = false;
98:     R2Status = true;
99:     return(ChR2 -> Get());
100: }
101:
102: // Method to add a new device to the channel's device table.
103: // Typically this is called from the T1410IODevice base class constructor
104:
105: void T1410Channel::AddIODevice(T1410IODevice *iodevice, int devicenumber) {
106:     assert(Devices[devicenumber] == NULL);
107:     Devices[devicenumber] = iodevice;
108: }
109:
110: // Channel is reset during ComputerReset
111:
112: void T1410Channel::OnComputerReset()
113: {
114:     Reset();
115: }
116:
117: void T1410Channel::Reset() {
118:     ChStatus = 0;
119:     R1Status = R2Status = false;
120:     MoveMode = LoadMode = false;
121:     ExtEndofTransfer = IntEndofTransfer = false;
122:     CycleRequired = false;
123:     InputRequest = false;
124:     LastInputCycle = false;
125:     EndofRecord = false;
126:     ChInterlock -> Reset();
127:     ChrBCInterlock -> Reset();
128:     ChRead -> Reset();
129:     ChWrite -> Reset();
130:     ChOverlap -> Reset();
131:     ChNotOverlap -> Reset();
132: }
```

```
133:
134: //  Channel is not reset during Program Reset
135:
136: void T1410Channel::OnProgramReset()
137: {
138:     //  Channel not affected by Program Reset
139: }
140:
141: //  Display Routine.
142:
143: void T1410Channel::Display() {
144:
145:     int i;
146:
147:     ChStatusDisplay[IOLAMPNOTREADY] -> Enabled =
148:         ((ChStatus & IOCHNOTREADY) != 0);
149:     ChStatusDisplay[IOLAMPBUSY] -> Enabled =
150:         ((ChStatus & IOCHBUSY) != 0);
151:     ChStatusDisplay[IOLAMPDATACHECK] -> Enabled =
152:         ((ChStatus & IOCHDATACHECK) != 0);
153:     ChStatusDisplay[IOLAMPCONDITION] -> Enabled =
154:         ((ChStatus & IOCHCONDITION) != 0);
155:     ChStatusDisplay[IOLAMPWLRECORD] -> Enabled =
156:         ((ChStatus & IOCHWLRECORD) != 0);
157:     ChStatusDisplay[IOLAMPNOTTRANSFER] -> Enabled =
158:         ((ChStatus & IOCHNOTTRANSFER) != 0);
159:
160:     for(i=0; i <= 5; ++i) {
161:         ChStatusDisplay[i] -> Repaint();
162:     }
163:
164:     //  Although in most instances the following would be redundant,
165:     //  because these objects are also on the CPU display list, we include
166:     //  them here in case we want to display a channel separately.
167:
168:     ChInterlock -> Display();
169:     ChRBCTInterlock -> Display();
170:     ChRead -> Display();
171:     ChWrite -> Display();
172:     ChOverlap -> Display();
173:     ChNotOverlap -> Display();
174: }
175:
176: //  Channel Lamp Test
177:
178: void T1410Channel::LampTest(bool b)
179: {
180:     int i;
181:
182:     //  Note, we don't have to do anything to the TDisplayLatch objects in
183:     //  the channel for lamp test.  They will take care of themselves on a
184:     //  lamp test.
185:
186:     if(!b) {
187:         for(i=0; i <= 5; ++i) {
188:             ChStatusDisplay[i] -> Enabled = true;
189:             ChStatusDisplay[i] -> Repaint();
190:         }
191:     }
192:     else {
193:         Display();
194:     }
195: }
196:
197: //  Channel Output to Device
198: //  Conditions on Entry: B data register (B_REG) contains character to output
```

```
199: // Channel Register 1 (E1 or F1) should be empty.
200:
201: void T1410Channel::DoOutput() {
202:
203:     // Back in the "bad old days" one could implement flowcharts using
204:     // GOTO statements. Here, instead, we fake it using a Finite State
205:     // Machine and a loop.
206:
207:     BCD tempbcd;
208:     int state = 1;
209:
210:     CPU -> CycleRing -> Set(this == CPU -> Channel[CHANNEL1] ? CYCLE_E : CYCLE_F);
211:
212:     while(true) {
213:         switch(state) {
214:
215:             // State 1: Initial entry. E1/F1 should be empty.
216:             // If we hit GMWM, no more readouts - we won't return until done.
217:             // Otherwise, load E1/F1 from B (with or without WM depending on whether
218:             // the I/O is Load or Move mode)
219:
220:             // The states came from the console output flowchart. hopefully,
221:             // they will work for other devices, as well.
222:
223:             case 1:
224:                 assert(!GetR1Status());
225:                 if(CPU -> B_Reg -> Get().TestGMWM()) {
226:                     IntEndofTransfer = true;
227:                     state = 5;
228:                 }
229:                 else {
230:                     tempbcd = CPU -> B_Reg -> Get();
231:                     if(MoveMode) {
232:                         tempbcd.ClearWM();
233:                         tempbcd.SetOddParity();
234:                         SetR1(tempbcd); bus
235:                     }
236:                     state = 4;
237:                 }
238:                 break;
239:
240:             // State 2 is the normal return, having sent the character out,
241:             // without an end of transfer.
242:
243:             case 2:
244:                 return;
245:
246:             // Case 3 in the console flow chart is a simple continuation
247:
248:             // State 4: The workhorse state. We have data in R1, R2 or
249:             // both. If R2 is empty, transfer R1 to R2. Normally we will
250:             // then just return. But, if this is the very last character,
251:             // (perhaps, if via a storage wrap condition, via state 5),
252:             // send the character out directly.
253:
254:             // If R2 is already full, or we are at end of transfer, then
255:             // send the character off to the device, and reset R2.
256:
257:             case 4:
258:                 // Is R2 currently empty? If so, fill it, and check for
259:                 // storage wrap. If, after filling R2, we are not at end
260:                 // of transfer, jump off to the next state.
261:                 if(!GetR2Status()) {
262:                     MoverR1R2();
263:                     if(CPU -> STAR -> Gate() == STORAGE - 1) {
264:                         IntEndofTransfer = true;
```

```
265:                     state = 5;
266:                     break;
267:                 }
268:                 if(!IntEndofTransfer) {
269:                     state = 2;
270:                     break;
271:                 }
272:             }
273:
274:             // If we get here, R2 has data, and either R1 is also full
275:             // or we are at end of transfer.  If we are at end of transfer,
276:             // handle that appropriately.
277:
278:             assert(GetR1Status() || IntEndofTransfer);
279:             CurrentDevice -> DoOutput();
280:             ResetR2();
281:             state = (IntEndofTransfer ? 5 : 4);
282:             break;
283:
284:             // Case 5: Internal end of transfer is set.  But we may still have
285:             // data in R1 or R2 to send off to the device.
286:
287:             case 5:
288:                 assert(IntEndofTransfer);
289:                 if(GetR1Status() || GetR2Status()) {
290:                     state = 4;                                // Off to send more data
291:                     break;
292:                 }
293:                 ExtEndofTransfer = true;
294:             return;
295:
296:             default:
297:                 assert(false);
298:                 break;
299:
300:             } // End Switch
301:         } // End loop
302:
303:         assert(false);
304:     }
305:
306: // Channel Input Processing (shared overlap/not overlap code)
307:
308: void T1410Channel::DoInput(TAddressRegister *addr) {
309:
310:     CycleRequired = false;                         // Reset cycle required
311:     InputRequest = false;                          // Reset co-routine flag
312:     *(CPU -> STAR) = *addr;                      // Copy memory address
313:     CPU -> Readout();                            // Get existing memory
314:
315:     CPU -> AChannel -> Select();                // Gate A channel appropriately
316:     this == CPU -> Channel[CHANNEL1] ?
317:         TACChannel::A_Channel_E : TACChannel::A_Channel_F ;
318:
319:     // If no data from device, the end is near...
320:
321:     if(!(GetR1Status() || GetR2Status())) {
322:         LastInputCycle = true;
323:     }
324:
325:     // If B GMWM, we are typically done storing data (unless this
326:     // is read to end of core, which ignores GMWM).  Unless, of course,
327:     // we have already hit end of record.
328:
329:     // Note that we can only check the op mod if we are NOT overlapped
330:     // (Note that the "to end of core" mods are all not overlapped)
```

```
331:
332:     if(!EndofRecord && CPU -> B_Reg -> Get().TestGMWM()) {
333:         if(ChOverlap -> State() || 
334:             (CPU -> Op_Mod_Reg -> Get().ToInt() & 0x3f) != OP_MOD_SYMBOL_DOLLAR) {
335:             EndofRecord = true;
336:         }
337:     }
338:
339:     // If no more input, or we hit GMWM, or memory just wrapped,
340:     // set internal end of transfer, but keep on accepting characters
341:     // until External end of transfer.
342:
343:     if(LastInputCycle || EndofRecord || CPU -> StorageWrapLatch) {
344:         IntEndofTransfer = true;
345:         if(!ExtEndofTransfer) {
346:             return;
347:         }
348:     }
349:
350:     // If we get here, we are either ending, or we are still storing data
351:
352:     if(!LastInputCycle && !IntEndofTransfer) { // Still storing input?
353:
354:         // If parity is no good, set data check...
355:
356:         if(!CPU -> AChannel -> Select().CheckParity()) {
357:             SetStatus(GetStatus() | IOCHDATACHECK);
358:
359:             // If asterisk insert, store an asterisk!
360:
361:             if(FI1415CE -> AsteriskInsert -> Checked) {
362:                 Chr2 -> Set(BCD_ASTERISK);
363:                 CPU -> AChannel -> Select(
364:                     this == CPU -> Channel[CHANNEL1] ?
365:                         TACchannel::A_Channel_E : TACchannel::A_Channel_F);
366:                 }
367:             } // End initial parity check
368:
369:             // Store the data (perhaps with a parity error!
370:
371:             CPU -> Store(CPU -> AssemblyChannel -> Select(
372:                 (MoveMode ?
373:                     TAssemblyChannel::AsmChannelWMB :
374:                     TAssemblyChannel::AsmChannelWMA),
375:                     TAssemblyChannel::AsmChannelZonesA,
376:                     false,
377:                     TAssemblyChannel::AsmChannelSignNone,
378:                     TAssemblyChannel::AsmChannelNumA) );
379:
380:             // If we have bad data, but not asterisk insert, STOP
381:             // One good way: Run FORTRAN w/o Asterisk Insert! ;)
382:
383:             if(!CPU -> AChannel -> Select().CheckParity()) {
384:                 CPU -> AChannelCheck -> SetStop("Data Check, No Asterisk Insert!");
385:                 return;
386:             }
387:
388:             ResetR2();                                // Reset Channel Data
389:             addr -> Set(CPU -> STARMod(1));          // Bump address register
390:
391:         } // End, !LastInputCycle
392:
393:         // In the real world, the device keeps reading data, and will
394:         // then set CycleRequired. (In fact, the console matrix will
395:         // call ChannelStrobe to actually do this when a key is pressed).
396:         // But most of our input devices in the emulator are co-routines,
```

```
397:     // so we have to call them back to give them a chance to strobe
398:     // the channel again. Also, in the real world, a device would
399:     // keep reading input data after the channel set Internal End of
400:     // Transfer -- we have to give the input co-routine a way to
401:     // finish reading it's record, even if we are no longer storing
402:     // it because we hit a GMWM.
403:
404:     CurrentDevice -> DoInput();
405:
406: }
407:
408: bool T1410Channel::ChannelStrobe(BCD ch) {
409:
410:     // If R1 has data already, move it to R2.
411:     // (In the emulator, that should probably never happen!)
412:
413:     if(GetR1Status()) {
414:         MoveR1R2();
415:     }
416:
417:     // Load R1, and copy to R2 if there is room in R2.
418:
419:     SetR1(ch);
420:     if(!GetR2Status()) {
421:         MoveR1R2();
422:     }
423:
424:     // If the channel has not already terminated the transfer,
425:     // ask to send the data to memory.
426:
427:     if(!IntEndofTransfer) {
428:         CycleRequired = true;
429:     }
430:
431:     InputRequest = true;                                // Force co-routine call
432:     return true;
433: }
434:
435: □
436:
437: // Class T1410IODevice implementation. This is an *abstract* base class,
438: // intended to be used to derive actual I/O devices
439:
440: T1410IODevice::T1410IODevice(int devicenumber, T1410Channel *Ch) {
441:     Channel = Ch;
442:     Ch -> AddIODevice(this,devicenumber);
443: }
444:
445: // And, finally, the 1410 IO Instruction Routines. We implement them
446: // here to keep the I/O stuff together!
447:
448: // Move and Load mode (M and L) Instructions.
449:
450: // Note: The channel selected by the CPU is known to be available, as the
451: // interlock test was passed during instruction readout at I3.
452: // IOChannelSelect indicates the selected channel.
453: // Channel -> ChUnitType has Device Type (e.g. 'T' for console)
454: // Channel -> ChUnitNumber has Unit Number
455:
456: void T1410CPU::InstructionIO() {
457:
458:     BCD opmod;
459:     T1410Channel *Ch = Channel[IOChannelSelect];
460:
461:     opmod = (Op_Mod_Reg -> Get() & 0x3f);
462:     assert(!(Ch -> ChInterlock -> State()));
```

```
463: // Reset the channel, then set Channel Interlock
464: Ch -> Reset();
465: Ch -> ChInterlock -> Set();
466:
467: // Set Move mode or Load mode, appropriately
468:
469: if(Op_Reg -> Get() & 0x3f) == OP_IO_MOVE) {
470:     Ch -> MoveMode = true;
471: }
472: else {
473:     Ch -> LoadMode = true;
474: }
475:
476:
477: // Start things out, depending on op modifier.
478:
479: switch(opmod.ToInt()) {
480:
481: case OP_MOD_SYMBOL_R:
482: case OP_MOD_SYMBOL_DOLLAR:
483:     Ch -> ChRead -> Set();
484:     break;
485:
486: case OP_MOD_SYMBOL_W:
487: case OP_MOD_SYMBOL_X:
488:     Ch -> ChWrite -> Set();
489:     break;
490:
491: default:
492:     InstructionCheck ->
493:         SetStop("Instruction Check: Invalid I/O d-character");
494:     return;
495: } // End switch on op modifier
496:
497: // See if there is a device for this device number. If not,
498: // return not ready.
499:
500:
501: if(Ch -> GetCurrentDevice() == NULL) {
502:     Ch -> SetStatus(IOCHNOTREADY);
503:     IRingControl = true;
504:     return;
505: }
506:
507: // Start up the I/O, do initial status check (Status Sample A)
508: // Just return if the status is not 0.
509:
510: Ch -> SetStatus(Ch -> GetCurrentDevice() -> Select());
511: if(Ch -> GetStatus() != 0) {
512:     IRingControl = true;
513:     return;
514: }
515:
516: // If OK so far, set Overlap/NotOverlap
517:
518: if(CPU -> IOOverlapSelect) {
519:     Ch -> ChOverlap -> Set();
520:     // TODO: Ch -> RequestOutput() or RequestInput (Write/Read)
521:     IRingControl = true;
522:     return;
523: }
524: else {
525:     Ch -> ChNotOverlap -> Set();
526: }
527:
528: // If we get here, we are not overlapped.
```

```
529:  
530:     if((opmod.ToInt() & OP_MOD_SYMBOL_W) == OP_MOD_SYMBOL_W) {  
531:         CPU -> Display();  
532:         while(!Ch -> ExtEndofTransfer) {  
533:             *STAR = *B_AR;  
534:             Readout();  
535:             Ch -> DoOutput();  
536:             B_AR -> Set(STARMod(1));  
537:         }  
538:         Ch -> ChNotOverlap -> Reset();  
539:         Ch -> SetStatus(Ch -> GetCurrentDevice() -> StatusSample());  
540:         IRingControl = true;  
541:         return;  
542:     }  
543:     else if((opmod.ToInt() & OP_MOD_SYMBOL_R) == OP_MOD_SYMBOL_R) {  
544:         // Input processing continues so long as not External End from device  
545:         CPU -> Display();  
546:         while(!Ch -> ExtEndofTransfer) {  
547:             // If no input, just wait here (not overlapped -- stuck here)  
548:             while(!(Ch -> CycleRequired || Ch -> InputRequest ||  
549:                   Ch -> ExtEndofTransfer)) {  
550:                 Application -> ProcessMessages();  
551:                 // sleep(10);  
552:                 continue; redundant  
553:             }  
554:             Ch -> DoInput(B_AR);  
555:         } // End, not External End of Transfer  
556:         assert(Ch -> ExtEndofTransfer);  
557:         // If, at the end, things do not match up ==> Wrong Length Record  
558:         // (Note that we do *not* test InputRequest here!)  
559:         if(Ch -> CycleRequired || Ch -> GetR2Status() || !Ch -> EndofRecord) {  
560:             Ch -> SetStatus(Ch -> GetStatus() | IOCHWLRECORD);  
561:         }  
562:         // And all done - continue with Instructions  
563:         Ch -> ChNotOverlap -> Reset();  
564:         IRingControl = true;  
565:         return;  
566:     }  
567: }
```

```

temp bcd = CPU → B-Reg → Get();
assert (!GetR1Status());
if (tempbcd, TestGMWU()) {
    Int End of Transfer = true;
    End of Read = true;
}
else if (MoreMode) {
    if (MoreMode) {
        temp bcd. Clear WM();
        temp bcd. Set Odd Parity();
    }
    SetR1(tempbcd);
}
if (StorageWrplatch) {
    Intend + Transfer = true;
    End of Read = true;
}

while (GetR1Status() || GetR2Status()) {
    if (GetR1Status() & !GetR2Status()) {
        Move R1R2();
    }
    Current Device → DoOutput();
    ResetR2();
}

if (Int End of Transfer) {
    Ext End of Transfer = true;
}

```

Display - Wrap - discs

31/7/11

31/7/11

~~①~~ started w/ E (missed 1<sup>st</sup> char.)

~~②~~ Displayed # (should not)

③ when done, still showed 24K  
WRITE } s/b clea.  
~~Not Overlays~~ }

break IN CNAME hydro.dynamicons

1. 102.00

2. 10,732

3. 6,9 " "

4. 0

5. 6,100 (2<sup>-4</sup>)

6. 4,103.2 (2<sup>-5</sup>)

(102.)

Page keeper

Keypss

Channel Stacks (hub-kes)

Data → R1  
→ R2 (rechts R1)

R0.11 cycle required if  
! intend of transfer

24,862.58

- 2,602.56

- 1,813.50

- 424.13

- 1311.74

18,710.65

3,000 .00 R1

15,710.65

2,000 .00 via

13,710.65

✓ UI1410 CNT.h

✓ IngRegLatch (Page 10)

✓ UI1410 CPT.h

✓ IngRegLatch = false

✓ UI1410 PWF.cpp

✓ CPU → IngRegLatch = false (2 times) - both routes

UI1415 IO.cpp

✓ Handler for inging request button

CPU → INQRegLatch = true;

INGRLSL → Enabled = true;  
Cancel → " "

Handler for inging cancel button

CPU → IngRegLatch = false;

INGRLSC → Enabled = false;  
IngCancel → " "

Handler for inging release button

CPU → IngRegLatch = false;

INGRLR → Enabled = false;  
IngCancel → " "

✓ UI1410 Branch.cpp

OP\_MOD\_SYMBOL\_Q (add to UI1410INST.h)

BranchLatch = IngRegLatch

✓ UI1415IO / T1415Console

(Remove) IOInProgress

UI1415IO

Matrix 3x8

Do 'Σ' print if reading

(key press handler)

✓ for CONSOLE - INPUT - L:

assert (GetMatrix == 32);  
DoMatrix();

Patru 32

```
else if (state == CONSOLE_INPUT_1) {  
    if (CPU → Channel[CHANNEL_1] ⇒ ChannelStrobe(bcd-key)) {  
        SendBCDTo1415(bcd-key);  
    }  
}
```

~~for  
bool~~ ✓ T1410Channel1::ChannelStrobe (BCD ch) {  
    if (GetR1Status()) {  
        return false; // No room!  
    }  
    Ch R1 → Set(ch);  
    if (GetR2Status()) {  
        MoveR1R2();  
    }  
    if (!IntEndOfTransfer) {  
        CycleRequired = true;  
    }  
    return true;  
}

✓ Channel Add  
    Cycle Required  
    Last Input Cycle  
    End of Reward

```
803: // 1415 Console Input/Ouptut (Channel) Device class implementation
804:
805: T1415Console::T1415Console(int devicenum, T1410Channel *Channel) :
806:     T1410IODevice(devicenum, Channel) {
807:
808:     IOInProgress = false;
809: }
810:
811: // Initial I/O Startup (also known as Status Sample A time)
812:
813: int T1415Console::Select() {
814:
815:     // If this is not channel 0, or not unit 0, return NOT READY
816:
817:     if(Channel -> GetUnitNumber() != 0 || Channel != CPU -> Channel[CHANNEL1]) {
818:         return(IOCNOTREADY);
819:     }
820:
821:     // If the console matrix is not at home, return BUSY
822:
823:     if(FI1415IO -> GetMatrix() != CONSOLE_MATRIX_HOME) {
824:         return(IOCCHBUSY);
825:     }
826:
827:     // Set up the console appropriately, depending on whether we are
828:     // reading or writing.
829:
830:     if(Channel -> ChWrite -> State()) {
831:         FI1415IO -> SetMatrix(30);
832:     }
833:     else {
834:         // Read is not implemented yet
835:     }
836:     return(0);
837: }
838:
839: // Handle an output request from the I/O instruction or Channel
840:
841: void T1415Console::DoOutput() {
842:
843:     // We might still be sending the "R". If so, finish that first.
844:
845:     while(FI1415IO -> GetMatrix() != 32) {
846:         FI1415IO -> DoMatrix();
847:         FI1415IO -> StepMatrix();
848:     }
849:
850:     // Send out a character.
851:
852:     FI1415IO -> DoMatrix();
853: }
854:
855: void T1415Console::DoInput() {
856:
857:     // Input is not implemented yet.
858: }
859:
860: // Finally, at the end of an operation, Finish things up, and check
861: // the final outcome.
862:
863: int T1415Console::StatusSample() {
864:
865:     FI1415IO -> SetMatrix(34);
866:     FI1415IO -> DoMatrix();
867:     FI1415IO -> ResetMatrix();
868:     return(0);
869: }
```

Selc. 1 C)

C if (! Clu → Inv Reg L+L) {  
return (IOCHNOTTRANSFER),  
}

FI1415 IO → SetMatrix (5, 6);

(30)

FI1415 TO → DoMatrix();

✓ FI1415TU → StepMatrix();

3

FI1415IV → DoMatrix();

FI1415ZO → StepMatrix();

FI1415ZU → SetState (CONSOLE-  
INPUT-1),  
NORMAL  
LOAD

(4)

①

```

if (!Opmod. ToInt() & OP-MOD-SYMBOL-R) == OP-MOD-SYMBOL-R) {
    while (!Ch → ExtEnd of Transfer.) {
        if (!Ch → CycleRequired()) {
            Application → Process Messages();
            Sleep (10); (ns!)
        } continue;
        *STAR = *B-AR; AChannel → Select (
        Readout(); Ch == Channel [CHANNEL 1] ?
        A-Channel-E : A-Channel-F );
        if (!GetR1Status() || GetR2Status ()) {
            Ch → Last Input Cycle = true;
        }
        if (B-Key → TestGMUM ());
            Ch → End of Record' = true;
        }
        if (Ch → Last Input Cycle || Ch → End of Record) {
            Ch → IntEnd of Transfer = true;
            TAssembly, Channel → Select (
            TAssembly, Channel !! AsmChannel (W.M.; B,
            TAssembly, Channel !! AsmChannel Zeros B,
            false,
            TAssembly, Channel !! AsmChannel Sign None
            TAssembly, Channel !! AsmChannel NumB );
            TAssembly, Channel !! AsmChannel NumB );
        }
        If (!ExtEnd of Transfer) {
            continue;
        }
    }
}
// Last Input or End of Record

```

~~make  
spurious~~

(2)

```

if (!Ch → Last Input Cycle) {
    if (!AChannel → Select(), Check Parity ()) {
        Ch → SetStatus(Ch → GetStatus() | IODATAEJECT);
        if (WI1415CE → Asterisk Insert → Checked) {
            Ch → ChR2 → Set(BCD-ASTERISK);
            AChannel → Select(
                Ch == Channel[CHANNEL1] ?
                    A-ChannelE, A-Channel-F );
        }
    }
}

```

```

Store (Assembly, Channel → Select (
    (Ch → MoveMode ?
        TAssembly, Channel :: AsmChannelWMB,
        TAssembly, Channel :: AsmChannel WMA),
    TAssembly, Channel :: AsmChannel Zones A,
    false,
    TAssembly, Channel :: AsmChannel Sign None,
    TAssembly, Channel :: AsmChannel NormA));
if (!AChannel → Select(), Check Parity ()) {
    AChannelCheck → SetStop ("Data Check");
    return;
}

```

```

B-AR = Set(STARMod(1));
}
// !Last Input Cyc

```

```

} // While Not Ext End At Xfr

```

$\Sigma$  ( \_\_\_\_\_ )

assert (ch  $\rightarrow$  Exit End of Transfer);

if ( ch  $\rightarrow$  Cycle Keywind ||

ch  $\rightarrow$  Get R2 Status() ||

| Ch  $\rightarrow$  EndOfRecord) {

Ch  $\rightarrow$  Set Status(Ch  $\rightarrow$  Get Status | IN.WLRECORD);

}

IRingControl = true;

}

```
389: // This class defines what is in an I/O Channel
390:
391: #define IOCHNOTREADY      1
392: #define IOCHBUSY          2
393: #define IOCHDATACHECK    4
394: #define IOCHCONDITION     8
395: #define IOCHNOTTRANSFER   16
396: #define IOCHWLRECORD      32
397:
398: #define IOLAMPNOTREADY    0
399: #define IOLAMPBUSY         1
400: #define IOLAMPDATACHECK   2
401: #define IOLAMPCONDITION    3
402: #define IOLAMPNOTTRANSFER  4
403: #define IOLAMPWLRECORD     5
404:
405: // Forward declaration of I/O Device class, which T1410Channel uses.
406:
407: class T1410IODevice;
408:
409: class T1410Channel : public TDisplayObject {
410:
411: public:
412:
413:     void Reset();
414:
415:     // Functions inherited from abstract base class classes now need definition
416:
417:     void OnComputerReset();
418:     void OnProgramReset();
419:     void Display();
420:     void LampTest(bool b);
421:
422: private:
423:
424:     // Channel information
425:
426:     int ChStatus;                                // Channel status (see defines)
427:     TLabel *ChStatusDisplay[6];                  // Channel status lights
428:     bool R1Status, R2Status;                     // State of R1, R2. True if full
429:     class T1410IODevice *Devices[64];            // Pointer to devices
430:
431: public:
432:
433:     TRegister *ChOp;
434:     TRegister *ChUnitType;
435:     TRegister *ChUnitNumber;
436:     TRegister *ChR1, *ChR2;
437:
438:     TDisplayLatch *ChInterlock;
439:     TDisplayLatch *ChRBCInterlock;
440:     TDisplayLatch *ChRead;
441:     TDisplayLatch *ChWrite;
442:     TDisplayLatch *ChOverlap;
443:     TDisplayLatch *ChNotOverlap;
444:
445:     bool MoveMode, LoadMode;                    // Mode: Without/With WM
446:     bool IntEndofTransfer, ExtEndofTransfer;    // Transfer end flags
447:     T1410IODevice *CurrentDevice;              // Ptr to device doing transfer
448:
449:     enum TapeDensity {
450:         DENSITY_200_556 = 0, DENSITY_200_800 = 1, DENSITY_556_800 = 2
451:     } TapeDensity;
452:
453:
454:     // Methods
```

```
455:  
456:     T1410Channel(                                // Constructor  
457:         TLabel *LampInterlock,  
458:         TLabel *LampRBCInterlock,  
459:         TLabel *LampRead,  
460:         TLabel *LampWRite,  
461:         TLabel *LampOverlap,  
462:         TLabel *LampNotOverlap,  
463:         TLabel *LampNotRead,  
464:         TLabel *LampBusy,  
465:         TLabel *LampDataCheck,  
466:         TLabel *LampCondition,  
467:         TLabel *LampWLRecord,  
468:         TLabel *LampNoTransfer  
469:     );  
470:  
471: // The following routine is called from T1410IODevice constructors  
472: // to add devices into the Channel's device table.  
473:  
474: void AddIODevice(T1410IODevice *iodevice, int devicenumber);  
475:  
476: // Methods that do most of the Channel's real work.  
477:  
478: inline int SetStatus(int i) { return ChStatus = i; }  
479: inline int GetStatus() { return ChStatus; }  
480: inline int GetDeviceNumber()  
481:     return ChUnitType -> Get().ToInt() & 0x3f;  
482:  
483: T1410IODevice *SetCurrentDevice()  
484:     return CurrentDevice = Devices[GetDeviceNumber()];  
485:  
486: T1410IODevice *GetCurrentDevice() { return CurrentDevice; }  
487: inline int GetUnitNumber()  
488:     return ChUnitNumber -> Get().ToAscii() - '0';  
489:  
490: void DoOutput();  
491:  
492: // Channel register methods  
493:  
494: inline bool GetR1Status() { return R1Status; }  
495: inline bool GetR2Status() { return R2Status; }  
496: inline void ResetR1() { R1Status = false; }  
497: inline void ResetR2() { R2Status = false; }  
498: BCD SetR1(BCD b);  
499: BCD SetR2(BCD b);  
500: BCD MoveR1R2();  
501: };  
502:  
503: // Class declaration for IO Devices. This is an *abstract* class,  
504: // which is used to derive a class for each kind of IO device (console,  
505: // reader, punch, tape, disk, etc.  
506:  
507: class T1410IODevice : public TObject {  
508:  
509: protected:  
510:     T1410Channel *Channel;                      // Channel device is attached to  
511:  
512: public:  
513:     T1410IODevice(int devicenum,T1410Channel *Channel); // Constructor  
514:     virtual int Select() = 0;                      // Start an operation  
515:     virtual int StatusSample() = 0;                // Return device status  
516:     virtual void DoOutput() = 0;                   // Channel -> Device  
517:     virtual void DoInput() = 0;                     // Device -> Channel  
518: };  
519:
```

```

1: //-----
2: #include <vc1.h>
3: #pragma hdrstop
4:
5: #include "UI1410CHANNEL.h"
6:
7: //-----
8: #pragma package(smart_init)
9:
10: #include <assert.h>
11: #include "UI1410CPUT.H"
12: #include "UI1410INST.H"
13:
14: // Implementation of I/O Channel Class
15:
16: // Constructor.  Initializes state
17:
18: T1410Channel::T1410Channel(
19:     TLabel *LampInterlock,
20:     TLabel *LampRBCInterlock,
21:     TLabel *LampRead,
22:     TLabel *LampWrite,
23:     TLabel *LampOverlap,
24:     TLabel *LampNotOverlap,
25:     TLabel *LampNotReady,
26:     TLabel *LampBusy,
27:     TLabel *LampDataCheck,
28:     TLabel *LampCondition,
29:     TLabel *LampWLRecord,
30:     TLabel *LampNoTransfer ) {
31:
32:     int i;
33:
34:     ChStatus = 0;
35:     TapeDensity = DENSITY_200_556;
36:     R1Status = R2Status = false;
37:
38:     ChStatusDisplay[IOLAMPNOTREADY] = LampNotReady;
39:     ChStatusDisplay[IOLAMPBUSY] = LampBusy;
40:     ChStatusDisplay[IOLAMPDATACHECK] = LampDataCheck;
41:     ChStatusDisplay[IOLAMPCONDITION] = LampCondition;
42:     ChStatusDisplay[IOLAMPNOTTRANSFER] = LampNoTransfer;
43:     ChStatusDisplay[IOLAMPWLRECORD] = LampWLRecord;
44:
45:     // Generally, the channel latches are *not* reset by Program Reset
46:
47:     ChInterlock = new TDisplayLatch(LampInterlock, false);
48:     ChrBCInterlock = new TDisplayLatch(LampRBCInterlock, false);
49:     ChRead = new TDisplayLatch(LampRead, false);
50:     ChWrite = new TDisplayLatch(LampWrite, false);
51:     ChOverlap = new TDisplayLatch(LampOverlap, false);
52:     ChNotOverlap = new TDisplayLatch(LampNotOverlap, false);
53:
54:     // Generally, the channel registers are *not* reset by Program Reset
55:
56:     ChOp = new TRegister(false);
57:     ChUnitType = new TRegister(false);
58:     ChUnitNumber = new TRegister(false);
59:     Chr1 = new TRegister(false);
60:     ChR2 = new TRegister(false);
61:
62:     // Clear the device table
63:
64:     for(i=0; i < 64; ++i) {
65:         Devices[i] = NULL;
66:     }

```

✓ *Cycle Required = false;  
 Last Input Cycle = false;  
 End of Record = false;*

```
67:
68:     CurrentDevice = NULL;
69:     MoveMode = LoadMode = false;
70:     ExtEndofTransfer = IntEndofTransfer = false;
71: }
72:
73: // Channel Register methods
74:
75: BCD T1410Channel::SetR1(BCD b) {
76:     ChR1 -> Set(b);
77:     R1Status = true;
78:     return(b);
79: }
80:
81: BCD T1410Channel::SetR2(BCD b) {
82:     ChR2 -> Set(b);
83:     R2Status = true;
84:     return(b);
85: }
86:
87: // Move data from register 1 to register 2
88: // Clearing register 1 in the process.
89:
90: BCD T1410Channel::MoveR1R2() {
91:     ChR2 -> Set(ChR1 -> Get());
92:     R1Status = false;
93:     R2Status = true;
94:     return(ChR2 -> Get());
95: }
96:
97: // Method to add a new device to the channel's device table.
98: // Typically this is called from the T1410IODevice base class constructor
99:
100: void T1410Channel::AddIODevice(T1410IODevice *iodevice, int devicenumber) {
101:     assert(Devices[devicenumber] == NULL);
102:     Devices[devicenumber] = iodevice;
103: }
104:
105: // Channel is reset during ComputerReset
106:
107: void T1410Channel::OnComputerReset()
108: {
109:     Reset();
110: }
111:
112: void T1410Channel::Reset() {
113:     ChStatus = 0;
114:     R1Status = R2Status = false;
115:     MoveMode = LoadMode = false;
116:     ExtEndofTransfer = IntEndofTransfer = false;
117:     ChInterlock -> Reset();
118:     ChRBCTInterlock -> Reset();
119:     ChRead -> Reset();
120:     ChWrite -> Reset();
121:     ChOverlap -> Reset();
122:     ChNotOverlap -> Reset();
123: }
124:
125: // Channel is not reset during Program Reset
126:
127: void T1410Channel::OnProgramReset()
128: {
129:     // Channel not affected by Program Reset
130: }
131:
132: // Display Routine.
```

CycleRequired = false;  
Last Input Cycle = false;  
End of Record = false;

```
133:
134: void T1410Channel::Display() {
135:
136:     int i;
137:
138:     ChStatusDisplay[IOLAMPNOTREADY] -> Enabled =
139:         ((ChStatus & IOCHNOTREADY) != 0);
140:     ChStatusDisplay[IOLAMPBUSY] -> Enabled =
141:         ((ChStatus & IOCHBUSY) != 0);
142:     ChStatusDisplay[IOLAMPDATACHECK] -> Enabled =
143:         ((ChStatus & IOCHDATACHECK) != 0);
144:     ChStatusDisplay[IOLAMPCCONDITION] -> Enabled =
145:         ((ChStatus & IOCHCONDITION) != 0);
146:     ChStatusDisplay[IOLAMPWLRECORD] -> Enabled =
147:         ((ChStatus & IOCHWLRECORD) != 0);
148:     ChStatusDisplay[IOLAMPNOTTRANSFER] -> Enabled =
149:         ((ChStatus & IOCHNOTTRANSFER) != 0);
150:
151:     for(i=0; i <= 5; ++i) {
152:         ChStatusDisplay[i] -> Repaint();
153:     }
154:
155: // Although in most instances the following would be redundant,
156: // because these objects are also on the CPU display list, we include
157: // them here in case we want to display a channel separately.
158:
159:     ChInterlock -> Display();
160:     ChRBCTerlock -> Display();
161:     ChRead -> Display();
162:     ChWrite -> Display();
163:     ChOverlap -> Display();
164:     ChNotOverlap -> Display();
165: }
166:
167: // Channel Lamp Test
168:
169: void T1410Channel::LampTest(bool b)
170: {
171:     int i;
172:
173: // Note, we don't have to do anything to the TDisplayLatch objects in
174: // the channel for lamp test. They will take care of themselves on a
175: // lamp test.
176:
177:     if(!b) {
178:         for(i=0; i <= 5; ++i) {
179:             ChStatusDisplay[i] -> Enabled = true;
180:             ChStatusDisplay[i] -> Repaint();
181:         }
182:     }
183:     else {
184:         Display();
185:     }
186: }
187:
188: // Channel Output to Device
189: // Conditions on Entry: B data register (B_REG) contains character to output
190: // Channel Register 1 (E1 or F1) should be empty.
191:
192: void T1410Channel::DoOutput() {
193:
194: // Back in the "bad old days" one could implement flowcharts using
195: // GOTO statements. Here, instead, we fake it using a Finite State
196: // Machine and a loop.
197:
198:     BCD tempbcd;
```

```
199:     int state = 1;
200:
201:     CPU -> CycleRing -> Set(this == CPU -> Channel[CHANNEL1] ? CYCLE_E : CYCLE_F);
202:
203:     while(true) {
204:         switch(state) {
205:
206:             // State 1: Initial entry. E1/F1 should be empty.
207:             // If we hit GMWM, no more readouts - we won't return until done.
208:             // Otherwise, load E1/F1 from B (with or without WM depending on whether
209:             // the I/O is Load or Move mode)
210:
211:             // The states came from the console output flowchart. hopefully,
212:             // they will work for other devices, as well.
213:
214:             case 1:
215:                 assert(!GetR1Status());
216:                 if(CPU -> B_Reg -> Get().TestGMWM()) {
217:                     IntEndofTransfer = true;
218:                     state = 5;
219:                 }
220:                 else {
221:                     tempbcd = CPU -> B_Reg -> Get();
222:                     if(MoveMode) {
223:                         tempbcd.ClearWM();
224:                         tempbcd.SetOddParity();
225:                         SetR1(tempbcd);
226:                     }
227:                     state = 4;
228:                 }
229:                 break;
230:
231:             // State 2 is the normal return, having sent the character out,
232:             // without an end of transfer.
233:
234:             case 2:
235:                 return;
236:
237:             // Case 3 in the console flow chart is a simple continuation
238:
239:             // State 4: The workhorse state. We have data in R1, R2 or
240:             // both. If R2 is empty, transfer R1 to R2. Normally we will
241:             // then just return. But, if this is the very last character,
242:             // (perhaps, if via a storage wrap condition, via state 5),
243:             // send the character out directly.
244:
245:             // If R2 is already full, or we are at end of transfer, then
246:             // send the character off to the device, and reset R2.
247:
248:             case 4:
249:                 // Is R2 currently empty? If so, fill it, and check for
250:                 // storage wrap. If, after filling R2, we are not at end
251:                 // of transfer, jump off to the next state.
252:                 if(!GetR2Status()) {
253:                     MoverR1R2();
254:                     if(CPU -> STAR -> Gate() == STORAGE - 1) {
255:                         IntEndofTransfer = true;
256:                         state = 5;
257:                         break;
258:                     }
259:                     if(!IntEndofTransfer) {
260:                         state = 2;
261:                         break;
262:                     }
263:                 }
264:
```

```
265:         // If we get here, R2 has data, and either R1 is also full
266:         // or we are at end of transfer.  If we are at end of transfer,
267:         // handle that appropriately.
268:
269:         assert(GetR1Status() || IntEndofTransfer);
270:         CurrentDevice -> DoOutput();
271:         ResetR2();
272:         state = (IntEndofTransfer ? 5 : 4);
273:         break;
274:
275:     // Case 5: Internal end of transfer is set.  But we may still have
276:     // data in R1 or R2 to send off to the device.
277:
278:     case 5:
279:         assert(IntEndofTransfer);
280:         if(GetR1Status() || GetR2Status()) {
281:             state = 4;                                // Off to send more data
282:             break;
283:         }
284:         ExtEndofTransfer = true;
285:         return;
286:
287:     default:
288:         assert(false);
289:         break;
290:
291:     } // End Switch
292: } // End loop
293:
294: assert(false);
295: }
296:
297: □
298:
299: // Class T1410IODevice implementation.  This is an *abstract* base class,
300: // intended to be used to derive actual I/O devices
301:
302: T1410IODevice::T1410IODevice(int devicenumber, T1410Channel *Ch) {
303:     Channel = Ch;
304:     Ch -> AddIODevice(this,devicenumber);
305: }
306:
307: // And, finally, the 1410 IO Instruction Routines.  We implement them
308: // here to keep the I/O stuff together!
309:
310: // Move and Load mode (M and L) Instructions.
311:
312: // Note: The channel selected by the CPU is known to be available, as the
313: // interlock test was passed during instruction readout at I3.
314: // IOChannelSelect indicates the selected channel.
315: // Channel -> ChUnitType has Device Type (e.g. 'T' for console)
316: // Channel -> ChUnitNumber has Unit Number
317:
318: void T1410CPU::InstructionIO() {
319:
320:     BCD opmod;
321:     T1410Channel *Ch = Channel[IOChannelSelect];
322:
323:     opmod = (Op_Mod_Reg -> Get() & 0x3f);
324:     assert(!(Ch -> ChInterlock -> State()));
325:
326:     // Reset the channel, then set Channel Interlock
327:
328:     Ch -> Reset();
329:     Ch -> ChInterlock -> Set();
330:
```

```
331:     // Set Move mode or Load mode, appropriately
332:
333:     if((Op_Reg -> Get() & 0x3f) == OP_IO_MOVE) {
334:         Ch -> MoveMode = true;
335:     }
336:     else {
337:         Ch -> LoadMode = true;
338:     }
339:
340:     // Start things out, depending on op modifier.
341:
342:     switch(opmod.ToInt()) {
343:
344:         case OP_MOD_SYMBOL_R:
345:         case OP_MOD_SYMBOL_DOLLAR:
346:             Ch -> ChRead -> Set();
347:             break;
348:
349:         case OP_MOD_SYMBOL_W:
350:         case OP_MOD_SYMBOL_X:
351:             Ch -> ChWrite -> Set();
352:             break;
353:
354:         default:
355:             InstructionCheck ->
356:                 SetStop("Instruction Check: Invalid I/O d-character");
357:             return;
358:     } // End switch on op modifier
359:
360:     // See if there is a device for this device number.  If not,
361:     // return not ready.
362:
363:     if(Ch -> GetCurrentDevice() == NULL) {
364:         Ch -> SetStatus(IOCHNOTREADY);
365:         IRingControl = true;
366:         return;
367:     }
368:
369:     // Start up the I/O, do initial status check (Status Sample A)
370:     // Just return if the status is not 0.
371:
372:     Ch -> SetStatus(Ch -> GetCurrentDevice() -> Select());
373:     if(Ch -> GetStatus() != 0) {
374:         IRingControl = true;
375:         return;
376:     }
377:
378:     // If OK so far, set Overlap/NotOverlap
379:
380:     if(CPU -> IOOverlapSelect) {
381:         Ch -> ChOverlap -> Set();
382:         // TODO: Ch -> RequestOutput() or RequestInput (Write/Read)
383:         IRingControl = true;
384:         return;
385:     }
386:     else {
387:         Ch -> ChNotOverlap -> Set();
388:     }
389:
390:     // If we get here, we are not overlapped.
391:
392:     if((opmod.ToInt() & OP_MOD_SYMBOL_W) == OP_MOD_SYMBOL_W) {
393:         while(!Ch -> ExtEndofTransfer) {
394:             *STAR = *B_AR;
395:             Readout();
396:             Ch -> DoOutput();
```

```
397:         B_AR -> Set(STARMod(1));
398:     }
399:     Ch -> ChNotOverlap -> Reset();
400:     Ch -> SetStatus(Ch -> GetCurrentDevice() -> StatusSample());
401:     IRingControl = true;
402:     return;
403: }
404: else {
405:     // Read not implemented yet.
406:     IRingControl = true;
407:     return;
408: }
409: }
```

## UI1410CHANNEL.cpp

```

351: void T1410CPU::InstructionIO() {
352:
353:     BCD opmod;
354:     T1410Channel *Ch = Channel[IOChannelSelect];
355:
356:     opmod = (Op_Mod_Reg -> Get() & 0x3f);
357:     assert(!(Ch -> ChInterlock -> State()));      *
358:
359:     // Reset the channel, then set Channel Interlock
360:
361:     Ch -> Reset();
362:     Ch -> ChInterlock -> Set();
363:
364:     // Set Move mode or Load mode, appropriately
365:
366:     if((Op_Reg -> Get() & 0x3f) == OP_IO_MOVE) {
367:         Ch -> MoveMode = true;
368:     }
369:     else {
370:         Ch -> LoadMode = true;
371:     }
372:
373:     // Start things out, depending on op modifier.
374:
375:     switch(opmod.ToInt()) {
376:
377:         case OP_MOD_SYMBOL_R:
378:         case OP_MOD_SYMBOL_DOLLAR:
379:             Ch -> ChRead -> Set();
380:             break;
381:
382:         case OP_MOD_SYMBOL_W:
383:         case OP_MOD_SYMBOL_X:
384:             Ch -> ChWrite -> Set();
385:             break;
386:
387:         default:
388:             InstructionCheck ->
389:                 SetStop("Instruction Check: Invalid I/O d-character");
390:
391:     } // End switch on op modifier
392:
393:     // See if there is a device for this device number. If not,
394:     // return not ready.
395:
396:     if(Ch -> GetCurrentDevice() == NULL) {
397:         Ch -> SetStatus(IOCHNOTREADY);
398:         IRingControl = true;
399:
400:     }
401:
402:     // Start up the I/O, do initial status check (Status Sample A)
403:     // Just return if the status is not 0.
404:
405:     Ch -> SetStatus(Ch -> GetCurrentDevice() -> Select());
406:     if(Ch -> GetStatus() != 0) {
407:         IRingControl = true;
408:
409:     }
410:
411:     // If OK so far, set Overlap/NotOverlap
412:
413:     if(CPU -> IOOverlapSelect) {
414:         Ch -> ChOverlap -> Set();
415:         // TODO: Ch -> RequestOutput() or RequestInput (Write/Read)
416:         IRingControl = true;

```

```

417:         return;
418:     }
419:     else {
420:         Ch -> ChNotOverlap -> Set();
421:     }
422:
423: // If we get here, we are not overlapped.
424:
425: if((opmod.ToInt() & OP_MOD_SYMBOL_W) == OP_MOD_SYMBOL_W) {
426:     while(!Ch -> ExtEndofTransfer) {
427:         *STAR = *B_AR;
428:         Readout();
429:         Ch -> DoOutput();
430:         B_AR -> Set(STARMod(1));
431:     }
432:     Ch -> ChNotOverlap -> Reset();
433:     Ch -> SetStatus(Ch -> GetCurrentDevice() -> StatusSample());
434:     IRingControl = true;
435:     return;
436: }
437: else if((opmod.ToInt() && OP_MOD_SYMBOL_R) == OP_MOD_SYMBOL_R) {
438:
439: // Input processing continues so long as not External End from device
440:
441:     while(!Ch -> ExtEndofTransfer) {
442:
443: // If no input, just wait here (not overlapped -- stuck here)
444: while !(Ch -> CycleRequired) { // Ch -> lastInputCycle
445:     Application -> ProcessMessages(); // ExtEnd of Transfer
446:     // sleep(10);
447:     continue;
448: }
449:
450:
451: { *STAR = *B_AR;
452:   Readout(); // Get existing B character
453:   AChannel -> Select(); // Gate A channel appropriately
454:   Ch == Channel[CHANNEL1] ?
455:       TACChannel::A_Channel_E : TACChannel::A_Channel_F );
456:
457: // If no data from device, the end is near...
458:
459: if(!(Ch -> GetR1Status() || Ch -> GetR2Status())) {
460:     Ch -> LastInputCycle = true;
461: }
462:
463: // If B GMWM, we are typically done storing data (unless this
464: // is read to end of core, which ignores GMWM)
465:
466: { if(opmod.ToInt() != OP_MOD_SYMBOL_DOLLAR && B_Reg -> Get().TestGMWM()) {
467:     Ch -> EndofRecord = true;
468: }
469:
470: // If no more input, or we hit GMWM, set internal end of
471: // transfer, but keep on accepting characters until
472: // External end of transfer.
473:
474: if(Ch -> LastInputCycle || Ch -> EndofRecord) {
475:     Ch -> IntEndofTransfer = true;
476:     if(!Ch -> ExtEndofTransfer) {
477:         continue;
478:     }
479: }
480:
481: // If we get here, we are either ending, or we are still
482: // storing data

```

```

483:             if(!Ch -> LastInputCycle) {           // Still storing input?
484:
485:                 // If parity is no good, set data check...
486:
487:                 if(!AChannel -> Select().CheckParity()) {
488:                     Ch -> SetStatus(Ch -> GetStatus() | IOCHDATACHECK);
489:
490:                     // If asterisk insert, store an asterisk!
491:
492:                     if(FI1415CE -> AsteriskInsert -> Checked) {
493:                         Ch -> Chr2 -> Set(BCD_ASTERISK);
494:                         AChannel -> Select(
495:                             Ch == Channel[CHANNEL1] ?
496:                                 TAChannel::A_Channel_E : TAChannel::A_Channel_F);
497:                         }
498:                     } // End initial parity check
499:
500:                     // Store the data (perhaps with a parity error!
501:
502:                     Store(AssemblyChannel -> Select(
503:                         (Ch -> MoveMode ?
504:                             TAssemblyChannel::AsmChannelWMB :
505:                             TAssemblyChannel::AsmChannelWMA),
506:                             TAssemblyChannel::AsmChannelZonesA,
507:                             false,
508:                             TAssemblyChannel::AsmChannelSignNone,
509:                             TAssemblyChannel::AsmChannelNumA) );
510:
511:
512:                     // If we have bad data, but not asterisk insert, STOP
513:                     // One good way: Run FORTRAN w/o Asterisk Insert! ;)
514:
515:                     if(!AChannel -> Select().CheckParity()) {
516:                         AChannelCheck -> SetStop("Data Check, No Asterisk Insert!");
517:                         return;
518:                     }
519:
520:                     Ch -> ResetR2();                   // Reset Channel Data
521:                     B_AR -> Set(STARMod(1));
522:                 } // End, !LastInputCycle
523:             } // End, not External End of Transfer
524:
525:             Ch -> ExtEndofTransfer;
526:             assert(Ch -> ExtEndofTransfer);
527:
528:             // If, at the end, things do not match up ==> Wrong Length Record
529:             if(Ch -> CycleRequired || Ch -> GetR2Status() || (!Ch -> EndofRecord)) {
530:                 Ch -> SetStatus(Ch -> GetStatus() | IOCHWLRECORD);
531:             }
532:
533:             // And all done - continue with Instructions
534:
535:             IRingControl = true;
536:             return;
537:         }
538:     }
539: }
```

*Temporary X*

```
305: bool T1410Channel::ChannelStrobe(BCD ch) {  
306:  
307:     // If R1 has data already, something went wrong!  
308:  
309:     if(GetR1Status()) {  
310:         return false; move R1R2()  
311:     }  
312:  
313:     // Load R1, and copy to R2 if there is room in R2.  
314:  
315:     chR1 → Set(ch); SetR1(ch);  
316:     if(GetR1Status()) {  
317:         • MoveR1R2();  
318:     }  
319:  
320:     // If the channel has not already terminated the transfer,  
321:     // ask to send the data to memory.  
322:  
323:     if(!IntEndofTransfer) {  
324:         CycleRequired = true;  
325:     }  
326:  
327:     return true;  
328: }
```

*Keywords →*

- interlock check fix VI1410INST.cpp Pg 10 (gAV Reset channel I3)
- remove JSMacModLatch, JSLoadModeLatch VI1410CPU.h Pg 16  
(they are per channel)
- Channel Add methods GetUnitType GetUnitNumber
- Define OP\_IOMOVE, OP\_IOWR
- VI1415IO inhibit printout if console not @ home (3/10 in progress)
- VI1415IO display R/W C matrix 3x8

---

3 methods  
WRITE  
NOT REPORT

```

389: // This class defines what is in an I/O Channel
390:
391: #define IOCHNOTREADY 1
392: #define IOCHBUSY 2
393: #define IOCHDATACHECK 4
394: #define IOCHCONDITION 8
395: #define IOCHNOTTRANSFER 16
396: #define IOCHWLRECORD 32
397:
398: #define IOLAMPNOTREADY 0
399: #define IOLAMPBUSY 1
400: #define IOLAMPDATACHECK 2
401: #define IOLAMPCONDITION 3
402: #define IOLAMPNOTTRANSFER 4
403: #define IOLAMPWLRECORD 5
404:
405: // Forward declaration of I/O Device class, which T1410Channel uses.
406:
407: class T1410IODevice;
408:
409: class T1410Channel : public TDisplayObject {
410:
411: public:
412:
413:     // Functions inherited from abstract base class classes now need definition
414:
415:     void OnComputerReset();
416:     void OnProgramReset();
417:     void Display();
418:     void LampTest(bool b);
419:
420: private:
421:
422:     // Channel information
423:
424:     int ChStatus;                                // Channel status (see defines)
425:     TLabel *ChStatusDisplay[6];                  // Channel status lights
426:     bool R1Status, R2Status;                      // State of R1, R2. True if full
427:     class T1410IODevice *Devices[64];            // Pointer to devices
428:
429: public:
430: ? TRegister *ChOp;                          ←
431: ? TRegister *ChUnitType;
432: ? TRegister *ChUnitNumber;
433: ? TRegister *ChR1, *ChR2;
434:
435:
436:     TDisplayLatch *ChInterlock;
437:     TDisplayLatch *ChRBCInterlock;
438:     TDisplayLatch *ChRead;
439:     TDisplayLatch *ChWrite;
440:     TDisplayLatch *ChOverlap;
441:     TDisplayLatch *ChNotOverlap;
442:
443:     enum TapeDensity {
444:         DENSITY_200_556 = 0, DENSITY_200_800 = 1, DENSITY_556_800 = 2
445:     } TapeDensity;
446:
447:
448:     // Methods
449:
450:     T1410Channel(                                     // Constructor
451:         TLabel *LampInterlock,
452:         TLabel *LampRBCInterlock,
453:         TLabel *LampRead,
454:         TLabel *LampWrite,

```

bool MacMode

bool Load Mode

bool Ext End of Transfer;

bool Int End of Transfers;

T1410 Device \*CurrentDevice;

```

455:     TLabel *LampOverlap,
456:     TLabel *LampNotOverlap,
457:     TLabel *LampNotRead,
458:     TLabel *LampBusy,
459:     TLabel *LampDataCheck,
460:     TLabel *LampCondition,
461:     TLabel *LampWLRecord,
462:     TLabel *LampNoTransfer
463: );
464:
465: // The following routine is called from T1410IODevice constructors
466: // to add devices into the Channel's device table.
467:
468: void AddIODevice(T1410IODevice *iodevice, int devicenumber);
469:
470: inline int SetStatus(int i) { return ChStatus = i; }
471: inline int GetStatus() { return ChStatus; }
472:
473: // Channel register methods
474:
475: inline bool GetR1Status() { return R1Status; }
476: inline bool GetR2Status() { return R2Status; }
477: inline void ResetR1() { R1Status = false; }
478: inline void ResetR2() { R2Status = false; }
479: BCD SetR1(BCD b);
480: BCD SetR2(BCD b);
481: BCD MoveR1R2();
482: };
483:
484: // Class declaration for IO Devices. This is an *abstract* class,
485: // which is used to derive a class for each kind of IO device (console,
486: // reader, punch, tape, disk, etc.
487:
488: class T1410IODevice : public TObject {
489:
490: protected:
491:     T1410Channel *Channel;           // Channel device is attached to
492:
493: public:
494:     T1410IODevice(int devicenum, T1410Channel *Channel); // Constructor
495:     virtual bool Select();          // Start an operation
496:     virtual BCD StatusSample();    // Return device status
497:     virtual Start(int unitnum, BCD op, BCD d); // Start an operation
498: };

```

*virtual int GetDeviceNum {  
return ChUnitType ~  
GetC().TUnit & 0x3f;*

*ditto Get Unit Number*

*Virtual DoOutput();*

*Virtual BCD StatusSample();*

*int*  
*Virtual void DoOutput()*  
*DoInput()*

```

1: //-----
2: #include <vcl.h>
3: #pragma hdrstop
4:
5: #include "UI1410CHANNEL.h"
6:
7: //-----
8: #pragma package(smart_init)
9:
10: #include <assert.h>
11: #include "UI1410CPUH.H"
12:
13: // Implementation of I/O Channel Class
14:
15: // Constructor. Initializes state
16:
17: T1410Channel::T1410Channel(
18:     TLabel *LampInterlock,
19:     TLabel *LampRBCInterlock,
20:     TLabel *LampRead,
21:     TLabel *LampWrite,
22:     TLabel *LampOverlap,
23:     TLabel *LampNotOverlap,
24:     TLabel *LampNotReady,
25:     TLabel *LampBusy,
26:     TLabel *LampDataCheck,
27:     TLabel *LampCondition,
28:     TLabel *LampWLRecord,
29:     TLabel *LampNoTransfer ) {
30:
31:     int i;
32:
33:     ChStatus = 0;
34:     TapeDensity = DENSITY_200_556;
35:     R1Status = R2Status = false;
36:
37:     ChStatusDisplay[IOLAMPNOTREADY] = LampNotReady;
38:     ChStatusDisplay[IOLAMPBUSY] = LampBusy;
39:     ChStatusDisplay[IOLAMPDATACHECK] = LampDataCheck;
40:     ChStatusDisplay[IOLAMPCONDITION] = LampCondition;
41:     ChStatusDisplay[IOLAMPNOTTRANSFER] = LampNoTransfer;
42:     ChStatusDisplay[IOLAMPWLRECORD] = LampWLRecord;
43:
44:     // Generally, the channel latches are *not* reset by Program Reset
45:
46:     ChInterlock = new TDisplayLatch(LampInterlock, false);
47:     ChrBCInterlock = new TDisplayLatch(LampRBCInterlock, false);
48:     ChRead = new TDisplayLatch(LampRead, false);
49:     ChWrite = new TDisplayLatch(LampWrite, false);
50:     ChOverlap = new TDisplayLatch(LampOverlap, false);
51:     ChNotOverlap = new TDisplayLatch(LampNotOverlap, false);
52:
53:     // Generally, the channel registers are *not* reset by Program Reset
54:
55:     ChOp = new TRegister(false);
56:     ChUnitType = new TRegister(false);
57:     ChUnitNumber = new TRegister(false);
58:     Chr1 = new TRegister(false);
59:     Chr2 = new TRegister(false);
60:
61:     // Clear the device table
62:
63:     for(i=0; i < 64; ++i) {
64:         Devices[i] = NULL;
65:     }
66: }

```

*Current Device = Null*

*MacMode = Low Mode = false;*  
*Extend of Transmission =*  
*IntEnd of Transmission =*  
*false;*

```

67:
68: // Channel Register methods
69:
70: BCD T1410Channel::SetR1(BCD b) {
71:     ChR1 -> Set(b);
72:     R1Status = true;
73:     return(b);
74: }
75:
76: BCD T1410Channel::SetR2(BCD b) {
77:     ChR2 -> Set(b);
78:     R2Status = true;
79:     return(b);
80: }
81:
82: // Move data from register 1 to register 2
83: // Clearing register 1 in the process.
84:
85: BCD T1410Channel::MoveR1R2() {
86:     ChR2 -> Set(ChR1 -> Get());
87:     R1Status = false;
88:     R2Status = true;
89:     return(ChR2 -> Get());
90: }
91:
92: // Method to add a new device to the channel's device table.
93: // Typically this is called from the T1410IODevice base class constructor
94:
95: void T1410Channel::AddIODevice(T1410IODevice *iodevice, int devicenumber) {
96:     assert(Devices[devicenumber] == NULL);
97:     Devices[devicenumber] = iodevice;
98: }
99:
100: // Channel is reset during ComputerReset
101: void T1410Channel::OnComputerReset()
102: {
103:     ChStatus = 0; R1Status = R2Status = false;
104:     MoveMode = LoadMode = false;
105:
106:     // Note: The objects which are TDisplayLatch objects will reset themselves
107: }
108:
109: // Channel is not reset during Program Reset
110:
111: void T1410Channel::OnProgramReset()
112: {
113:     // Channel not affected by Program Reset
114: }
115:
116: // Display Routine.
117:
118: void T1410Channel::Display() {
119:
120:     int i;
121:
122:     ChStatusDisplay[IOLAMPNOTREADY] -> Enabled =
123:         ((ChStatus & IOCHNOTREADY) != 0);
124:     ChStatusDisplay[IOLAMPBUSY] -> Enabled =
125:         ((ChStatus & IOCHBUSY) != 0);
126:     ChStatusDisplay[IOLAMPDATACHECK] -> Enabled =
127:         ((ChStatus & IOCHDATACHECK) != 0);
128:     ChStatusDisplay[IOLAMPCONDITION] -> Enabled =
129:         ((ChStatus & IOCHCONDITION) != 0);
130:     ChStatusDisplay[IOLAMPWLRECORD] -> Enabled =
131:         ((ChStatus & IOCHWLRECORD) != 0);
132:     ChStatusDisplay[IOLAMPNOTTRANSFER] -> Enabled =

```

*CurrentDevice = NULL;*  
*ExtEndofTransmfer =*  
*IntEndofTransmfer =*  
*~~ext, int~~; false*

```
133:     ((ChStatus & IOCHNOTTRANSFER) != 0);
134:
135:     for(i=0; i <= 5; ++i) {
136:         ChStatusDisplay[i] -> Repaint();
137:     }
138:
139: // Although in most instances the following would be redundant,
140: // because these objects are also on the CPU display list, we include
141: // them here in case we want to display a channel separately.
142:
143:     ChInterlock -> Display();
144:     ChRBCTInterlock -> Display();
145:     ChRead -> Display();
146:     ChWrite -> Display();
147:     ChOverlap -> Display();
148:     ChNotOverlap -> Display();
149: }
150:
151: // Channel Lamp Test
152:
153: void T1410Channel::LampTest(bool b)
154: {
155:     int i;
156:
157: // Note, we don't have to do anything to the TDisplayLatch objects in
158: // the channel for lamp test. They will take care of themselves on a
159: // lamp test.
160:
161:     if(!b) {
162:         for(i=0; i <= 5; ++i) {
163:             ChStatusDisplay[i] -> Enabled = true;
164:             ChStatusDisplay[i] -> Repaint();
165:         }
166:     }
167:     else {
168:         Display();
169:     }
170: }
171:
172: □
173:
174: // Class T1410IODevice implementation. This is an *abstract* base class,
175: // intended to be used to derive actual I/O devices
176:
177: T1410IODevice::T1410IODevice(int devicenumber, T1410Channel *Ch) {
178:     Channel = Ch;
179:     Ch -> AddIODevice(this,devicenumber);
180: }
181:
```

T1410 Channel :: DoOutput()

{

    bcd tempbcd;  
    int state = 1;

    CycleRing  $\rightarrow$  Set(Cycle-E);

    while(true) {

        switch state:

            case 1:

                if (B-Reg  $\rightarrow$  Get(), TestGMWM()) {

                    Int End of Transfer = true;

                    state = 5;

                }

            else {

                tempbcd = B-Reg  $\rightarrow$  Get();

                if (ModeMod) {

                    tempbcd = 01corWMC;

                    tempbcd. SetOdd Parity();

                    SetR1(tempbcd);

                }

                state = 4;

            }

            break;

        Case 2:

    return;

case 4:

```
if (!GetR2Status()) {  
    MockR2();  
    if (STAR > GetC() == STORAGE - 1) {  
        IntEndOfTransfer = true;  
        state = 5;  
        break;  
    }  
    if (!IntEndOfTransfer) {  
        state = 2;  
        break;  
    }  
}  
// Fall Thru if internal end (last char in R2)  
(CurrentDevice → DoOutput());  
ResetR2();  
state = (IntEndOfTransfer ? 5 : 4);  
break;
```

case 5:

```
if (GetR1Status() || GetR2Status()) {  
    state = 4;  
    break;  
}  
ExtEndOfTransfer = true;  
return;
```

else fault:

```
assert(false);  
break;
```

3

3

class T1415Console : public T1410 Device {

virtual bool Select();

virtual BCD Status SampleB();

T1415 Console (int deviceNum, T1410 Channel \*Channel);

bool ConsoleIOInProgress;

virtual void DoOutput()  
Input()

```
T1415Console::T1415Console(int devnum, T1410Channel *channel) :  
    T1410Device(devnum, Channel) {  
  
    ConsoleIOPInProgress = false;  
}
```

---

```
int T1415Console::Select() {  
    if (Channel  $\rightarrow$  ChUnitNumber != 0 || Channel  $\rightarrow$  Channel[CHANNEL]) {  
        return (IOCHANNOTREADY);  
    }  
    if (FI1415FO  $\rightarrow$  GetMatrix() != CONSOLE-MATRIX-HOME) {  
        return (IOCHTBUSY);  
    }  
    if (Channel  $\rightarrow$  ChUnit  $\rightarrow$  State()) {  
        FI1415FO  $\rightarrow$  SetMatrix(38);  
    }  
}
```

}

C

~~Introduzione cont.~~

T1415Console :: DoOutput()

{

while (FILE15IO → GetMatrix != 32) {

DoMatrix();

StepMatrix();

}

DoMatrix();

}

---

int T1415Console:: StepSample()

{

StepMatrix();

DoMatrix();

return (0);

}

new T1415 words (odessa run, Chisel)

Instruction IO() {
 Ch = Channel[IOChannelSelect];
 BCD opmod;
 // Note: The Channel is known to be available, as the
 // interlock check was passed at I3.
 // IO Channel Select indicates the selected Channel
 // Channel → ChUnitType has Unit type (device)
 // Channel → ChUnitNumber has Unit #
 Ch → ChInterlock → Set();
 if (Op-Reg → Get() == OP\_ID\_MOVE) {
 Ch → MoveMode = true;
 } else {
 Ch → LoadMode = true;
 }
 if (Ch → ChStatus != 0) { // Reset State.
 if (Ch → Device[Ch → GetUnitType] == NULL) {
 IOCHANREADY;
 }
 CurrentDevice Ch → ChStatus = 0;
 IfingControl = true;
 return;
 }
 // Do initial status check. If not clear, we are done
 CurrentDevice
 Ch → ChStatus = ... → Select();
 if (Ch → ChStatus != 0) {
 IfingControl = true;
 return;
 }
 opmod = (Op-Mod → Get() & OpMod);
 switch opmod {
 case OP-MOD-SYMBOL-R:
 case OP-MOD-SYMBOL-DOOR:
 Ch → ChWrite → Reset();
 Ch → ChRead → Set();
 }
 }

## Instruction IO P(2)

```
case OP-MOD-SYMBOL-W:  
case OP-MOD-SYMBOL-X:  
    Ch → ChRead → Read();  
    Ch → ChWrite → Set();
```

default:

```
    InstructionCheck →  
        setStop("Invalid I/O d-character.");  
    return;
```

end[1]

```
} if (Op-Reg → Get() & BITA) {
```

```
    Ch → ChOverlap → Reset();  
    Ch → ChNotOverlap → Set();
```

```
} else {
```

```
    Ch → ChOverlap → Set();  
    Ch → ChNotOverlap → Reset();  
    Ch → RequestOutput();  
    IRingControl = true;  
    return;
```

```
}
```

## Instruction IV

Not Overlapped

```
C if (opmod & OP_MOD_SYMBOL == OP_MOD_SYMBOL_W) {  
    while (!ch → Ext End of Transfer) {  
        *STAR = *B_AR;  
        Read ch();  
        ch → Do Output();  
    }  
    IRing Control = true;  
}
```

```
1: //-----
2: #include <vcl.h>
3: #pragma hdrstop
4:
5: #include "CHANNEL.h"
6:
7: //-----
8: #pragma package(smart_init)
9:
10: #include <stdio.h>
11: #include "UIMULTI.H"  !
12:
13: TChannel::TChannel() {
14:     int i;
15:
16:     for(i=0; i<64; ++i) {
17:         devices[i] = NULL;
18:     }
19: }
20:
21: void TChannel::Addio(TIO *iodevice, int device) {
22:     devices[device] = iodevice;
23: }
24:
25: bool TChannel::Startio(int device,int unit) {
26:
27:     char msg[80];
28:
29:     if(devices[device] == NULL) {
30:         sprintf(msg,"Bad device: %d",device);
31:         Testout -> DebugOut -> Lines -> Add(msg);
32:         return(false);
33:     }
34:     else {
35:         return devices[device] -> Startio(unit);
36:     }
37: }
38:
39: void TChannel::Ident() {
40:     Testout -> DebugOut -> Lines -> Add("The channel is here...");
41: }
42:
43: TIO::TIO(int device, char *n, TChannel *ch) {
44:     strcpy(name,n);
45:     Channel = ch;
46:     ch -> Addio(this, device);           // Add self to channel
47: }
48:
49: TestDevice::TestDevice(int device, char *n, TChannel *ch, char *m) :
50:     TIO(device,n,ch)                  // Base class constructor
51: {
52:     msg = m;
53: }
54:
55: bool TestDevice::Startio(int unit) {
56:     char msg[256];
57:
58:     sprintf(msg,"I/O Started for device %s, unit %d: %s",name,unit,msg);
59:     Testout -> DebugOut -> Lines -> Add(msg);
60:     return(true);
61: }
62:
```

```

1: //-----
2: #ifndef UIMULTIH
3: #define UIMULTIH
4: //-----
5: #include <Classes.hpp>
6: #include <Controls.hpp>
7: #include <StdCtrls.hpp>
8: #include <Forms.hpp>
9: //-----
10:
11: #include "SCANNER.H"
12:
13: class TTestout : public TForm
14: {
15:     __published:    // IDE-managed Components
16:         TMemo *DebugOut;
17:         TButton *Quit;
18:         TButton *Speak;
19:         void __fastcall QuitClick(TObject *Sender);
20:         void __fastcall SpeakClick(TObject *Sender);
21:     private:    // User declarations
22:
23:         TChannel *Channel;
24:         TestDevice *Reader;
25:         TestDevice *Punch;
26:         int device;
27:         int unit;
28:
29:     public:    // User declarations
30:         __fastcall TTestout(TComponent* Owner);
31:     };
32: //-----
33: extern PACKAGE TTestout *Testout;
34: //-----
35: #endif
36:

```

// Virtual

IDevice

Select (unit, R/W/ctrl)  
 Status Sample()  
 Start Read()  
 Start write()  
 Start Ctrl()

TChannel \* Channel

Channel

AddIO (IDevice \*, int device)

```
1: //-----
2: #include <vcl.h>
3: #pragma hdrstop
4:
5: #include "UIMULTI.h"
6: //-----
7: #pragma package(smart_init)
8: #pragma resource "*.*dfm"
9: TTestout *Testout;
10: //-----
11: __fastcall TTestout::TTestout(TComponent* Owner)
12:   : TForm(Owner)
13: {
14:   Channel = new TChannel();
15:   Reader = new TestDevice(10,"reader",Channel,"I am a card reader.");
16:   Punch = new TestDevice(11,"punch",Channel,"I am a card punch.");
17:   unit = 0;
18:   device = 9;
19: }
20: //-----
21: void __fastcall TTestout::QuitClick(TObject *Sender)
22: {
23:   Close();
24: }
25: //-----
26: void __fastcall TTestout::SpeakClick(TObject *Sender)
27: {
28:   DebugOut -> Lines -> Add("Here I am...");
29:   ++device;
30:   if(device > 12) {           // Yes, I know, 12 is > 11 ==> INTENTIONAL
31:     device = 10;
32:   }
33:   ++unit;
34:   unit = unit % 10;
35:   Channel -> Startio(device,unit);           // Use virtual function..
36: }
37: //-----
38:
```

```
1: //-----
2: #ifndef CHANNELH
3: #define CHANNELH
4: //-----
5:
6: class TIO;
7:
8: class TChannel : public TObject {
9:
10: private:
11:     TIO *devices[64];
12:
13: public:
14:     TChannel();                                // Constructor
15:     bool Startio(int device, int unit);        // Start an I/O operation
16:     void Addio(TIO *iodevice, int device);      // Add an I/O device
17:     void Ident();                             // Callback test from device
18: };
19:
20: class TIO : public TObject {
21:
22: protected:
23:     char name[32];                           // Device name
24:     TChannel *Channel;                      // Ptr to parent channel
25:
26: public:
27:     TIO(int device, char *name, TChannel *Channel); // Constructor
28:     virtual bool Startio(int unit) = 0;           // Each one has different I/O
29: };
30:
31: class TestDevice : public TIO {
32:
33: private:
34:     char *msg;                               // Differentiate
35:
36: public:
37:     TestDevice(int device, char *name, TChannel *Channel, char *msg); // Differentiate
38:     virtual bool Startio(int unit);           // Instantiate
39: };
40:
41: #endif
42:
```

T1410 channel : T1410channel  
constructor - Mer out Device

AddIODevice(T1410Device \*ioDevice, int deviceNumber);

T1410Device

Contr. (deviceNum, ^channel)  
Constructor  
calls AddIODevice  
inits channel

Class TIO {

private:  
TIO (char \*name, ~~int unit~~, TChannel \*Channel);  
char name [32];  
~~int unit;~~  
TChannel \*Channel;

public:

bool start (int unit);

} ;

class TChannel {

private:  
TChannel(); ✓  
TIO \*devices [64];  
~~bool reset();~~  
bool startio (int device, int unit);  
Addio (TIO \*ioDev, int device); ✓  
}; iden();

```
TIO::TIo (char *n , int , TChannel *Chanell) {  
    strcpy (name, n);  
    unit = u;  
    Channel = Ch;  
}
```

```
bool startio (int u){  
    Ch → ident();  
    fprintf (stder, "Io on device %s unit %d started.\n",  
            name, u);  
}
```

```
void TChannel::TChannel() {
    int i;
    for(i=0; i<64; ++i) {
        devices[i] = NULL;
    }
}
```

```
bool TChannel::addio(TIO *iodev, int device) {
    devices[device] = iodev;
}
```

```
bool TChannel::startio(int deviu, int unit) {
    if(devices[device] == NULL) {
        fprintf(stderr, "Device %d unit %d NOT RE ADD.\n",
                device, unit);
    }
    return devices[device] → start(unit);
}
```

```
bool TChannel::ident() {
    fprintf(stderr, "I am a channel\n");
}
```

INSTRUCTION      LOAD ENDS

J	dchar	9, @R	- Printer Cartridge
J	dchar	Q	- Inquiry
J	dchar	K	- Tape Mark
R, X	Channel wait co-routine		

```
1: //-----
2: #ifndef UI1415LH
3: #define UI1415LH
4: //-----
5: #include <vcl\Classes.hpp>
6: #include <vcl\Controls.hpp>
7: #include <vcl\StdCtrls.hpp>
8: #include <vcl\Forms.hpp>
9: #include <vcl\ComCtrls.hpp>
10: #include <vcl\ExtCtrls.hpp>
11: //-----
12:
13: // 1415 Light panel display
14:
15: class TF1415L : public TForm
16: {
17:     __published:    // IDE-managed Components
18:     TPageControl *T1415Display;
19:     TTabSheet *TSCPStatus;
20:     TTabSheet *TSIOChannels;
21:     TTabSheet *TSSystemCheck;
22:     TTabSheet *TSPowerSystemControls;
23:     TPanel *PCPU;
24:     TPanel *PStatus;
25:     TLabel *LabelCPU;
26:     TPanel *PIRing;
27:     TLabel *Light_I_OP;
28:     TLabel *Light_I_1;
29:     TLabel *Light_I_2;
30:     TLabel *Light_I_3;
31:     TLabel *Light_I_4;
32:     TLabel *Light_I_5;
33:     TLabel *Light_I_6;
34:     TLabel *Light_I_7;
35:     TLabel *Light_I_8;
36:     TLabel *Light_I_9;
37:     TLabel *Light_I_10;
38:     TLabel *Light_I_11;
39:     TLabel *Light_I_12;
40:     TLabel *LabelIRing;
41:     TPanel *PAring;
42:     TLabel *LabelARing;
43:     TLabel *Light_A_1;
44:     TLabel *Light_A_2;
45:     TLabel *Light_A_3;
46:     TLabel *Light_A_4;
47:     TLabel *Light_A_5;
48:     TLabel *Light_A_6;
49:     TPanel *PClock;
50:     TLabel *LabelClock;
51:     TLabel *Light_Clk_A;
52:     TLabel *Light_Clk_B;
53:     TLabel *Light_Clk_C;
54:     TLabel *Light_Clk_D;
55:     TLabel *Light_Clk_E;
56:     TLabel *Light_Clk_F;
57:     TLabel *Light_Clk_G;
58:     TLabel *Light_Clk_H;
59:     TLabel *Light_Clk_J;
60:     TLabel *Light_Clk_K;
61:     TPanel *PScan;
62:     TLabel *LabelScan;
63:     TLabel *Light_Scan_N;
64:     TLabel *Light_Scan_1;
65:     TLabel *Light_Scan_2;
66:     TLabel *Light_Scan_3;
```

```
67:     TPanel *PSubScan;
68:     TLabel *LabelSubScan;
69:     TLabel *Light_Sub_Scan_U;
70:     TLabel *Light_Sub_Scan_B;
71:     TLabel *Light_Sub_Scan_E;
72:     TLabel *Light_Sub_Scan_MQ;
73:     TPanel *PCycle;
74:     TLabel *LabelCycle;
75:     TLabel *Light_Cycle_A;
76:     TLabel *Light_Cycle_B;
77:     TLabel *Light_Cycle_C;
78:     TLabel *Light_Cycle_D;
79:     TLabel *Light_Cycle_E;
80:     TLabel *Light_Cycle_F;
81:     TLabel *Light_Cycle_I;
82:     TLabel *Light_Cycle_X;
83:     TPanel *PArith;
84:     TLabel *LabelArith;
85:     TLabel *Light_Carry_In;
86:     TLabel *Light_Carry_Out;
87:     TLabel *Light_A_Complement;
88:     TLabel *Light_B_Complement;
89:     TLabel *LabelStatus;
90:     TLabel *Light_B_GT_A;
91:     TLabel *Light_B_EQ_A;
92:     TLabel *Light_B_LT_A;
93:     TLabel *Light_Overflow;
94:     TLabel *Light_Divide_Overflow;
95:     TLabel *Light_Zero_Balance;
96:     TPanel *PIOCh;
97:     TPanel *PIOCh1Status;
98:     TLabel *LabelCh1Status;
99:     TLabel *Light_Ch1_NotReady;
100:    TLabel *Light_Ch1_Busy;
101:    TLabel *Light_Ch1_DataCheck;
102:    TLabel *Light_Ch1_Condition;
103:    TLabel *Light_Ch1_WLRecord;
104:    TLabel *Light_Ch1_NoTransfer;
105:    TPanel *PIOCh1Control;
106:    TLabel *LabelCh1Control;
107:    TLabel *Light_Ch1_Interlock;
108:    TLabel *Light_Ch1_RBCInterlock;
109:    TLabel *Light_Ch1_Read;
110:    TLabel *Light_Ch1_Write;
111:    TLabel *Light_Ch1_Overlap;
112:    TLabel *Light_Ch1_NoOverlap;
113:    TPanel *PIOCh2Control;
114:    TLabel *LabelCh2Control;
115:    TLabel *Light_Ch2_Interlock;
116:    TLabel *Light_Ch2_RBCInterlock;
117:    TLabel *Light_Ch2_Read;
118:    TLabel *Light_Ch2_Write;
119:    TLabel *Light_Ch2_Overlap;
120:    TLabel *Light_Ch2_NoOverlap;
121:    TPanel *PIOChControl;
122:    TPanel *PIOChStatus;
123:    TPanel *PIOCh2Status;
124:    TLabel *LabelCh2Status;
125:    TLabel *Light_Ch2_NotReady;
126:    TLabel *Light_Ch2_Busy;
127:    TLabel *Light_Ch2_DataCheck;
128:    TLabel *Light_Ch2_Condition;
129:    TLabel *Light_Ch2_WLRecord;
130:    TLabel *Light_Ch2_NoTransfer;
131:
132:    TPanel *Panell;
```

```
133:     TPanel *PProcess;
134:     TLabel *LabelProcess;
135:     TLabel *Light_Check_AChannel;
136:     TLabel *Light_Check_BChannel;
137:     TLabel *Light_Check_AssemblyChannel;
138:     TLabel *Light_Check_AddressChannel;
139:     TLabel *Light_Check_AddressExit;
140:     TLabel *Light_Check_ARegisterSet;
141:     TPanel *PProgram;
142:     TLabel *LabelProgram;
143:     TLabel *Light_Check_IOInterlock;
144:     TLabel *Light_Check_AddressCheck;
145:     TLabel *Light_Check_RBCInterlock;
146:     TLabel *Light_Check_InstructionCheck;
147:     TPanel *PSystemCheck;
148:     TLabel *Light_Check_BRegisterSet;
149:     TLabel *Light_Check_OpRegisterSet;
150:     TLabel *Light_Check_OpModifierSet;
151:     TLabel *Light_Check_ACharacterSelect;
152:     TLabel *Light_Check_BCharacterSelect;
153:     TPanel *LabelPower;
154:     TPanel *PPower;
155:     TLabel *Light_Thermal;
156:     TLabel *Light_CB_Trip;
157:     TLabel *Light_IO_Offline;
158:     TLabel *Light_Tape_Offline;
159:     TLabel *Light_Disk_Offline;
160:     TPanel *Label_SystemControls;
161:     TPanel *P_System_Controls;
162:     TLabel *Light_1401_Compat;
163:     TLabel *Light_Priority_Alert;
164:     TLabel *Light_Off_Normal;
165:     TLabel *Light_Stop;
166: private: // User declarations
167: public: // User declarations
168:     __fastcall TF1415L(TComponent* Owner);
169: };
170: //-----
171: extern TF1415L *F1415L;
172: //-----
173: #endif
```

```
1: // This Unit defines the behavior of the 1410 Light status display
2:
3: //-----
4: #include <vcl\vcl.h>
5: #pragma hdrstop
6:
7: #include "UI1415L.h"
8: //-----
9: #pragma resource "* .dfm"
10: TF1415L *F1415L;
11: //-----
12: __fastcall TF1415L::TF1415L(TComponent* Owner)
13:   : TForm(Owner)
14: {
15:   WindowState = wsMinimized;
16: }
17: //-----
```

```
1: //-----
2: #ifndef UI1415IOH
3: #define UI1415IOH
4: //-----
5: #include <vcl\Classes.hpp>
6: #include <vcl\Controls.hpp>
7: #include <vcl\StdCtrls.hpp>
8: #include <vcl\Forms.hpp>
9: //-----
10:
11: // 1415 Console User Interface
12:
13: #include "ubcd.h"
14: #include "UI1410CPUT.h"
15:
16: #include <vcl\ExtCtrls.hpp>
17:
18: #define CONSOLE_IDLE    1      // Console is idle
19: #define CONSOLE_NORMAL   2      // Normal Read Mode input
20: #define CONSOLE_LOAD     3      // Normal Load Mode input
21: #define CONSOLE_ALTER    4      // Console is in alter mode loading memory
22: #define CONSOLE_ADDR     5      // Console is in address set or display mode
23: #define CONSOLE_DISPLAY   6      // Console is displaying register or mem
24:
25: #define CONSOLE_MATRIX_HOME 0 // Console matrix home position
26:
27: class TFI1415IO : public TForm
28: {
29:     __published: // IDE-managed Components
30:         TMemo *I1415IO;
31:         TLabel *KeyboardLock;
32:         TButton *InqReq;
33:         TButton *InqRlse;
34:         TButton *InqCancel;
35:         TButton *WordMark;
36:         TButton *MarginRelease;
37:         TTimer *KeyboardLockReset;
38:         void __fastcall I1415IOKeyPress(TObject *Sender, char &Key);
39:
40:         void __fastcall I1415IOKeyDown(TObject *Sender, WORD &Key, TShiftState Shift);
41:
42:         void __fastcall WordMarkClick(TObject *Sender);
43:         void __fastcall MarginReleaseClick(TObject *Sender);
44:
45:         void __fastcall KeyboardLockResetTimer(TObject *Sender);
46: private: // User declarations
47:     int state;
48:     int matrix;
49:     bool WMCtrlLatch;
50:     bool DisplayWMCtrlLatch;
51:     bool DisplayFullLineLatch;
52:     TAddressRegister *Console_AR;
53:     void DoWordMark();
54:     void LockKeyboard();
55:     void UnlockKeyboard();
56: public: // User declarations
57:     __fastcall TFI1415IO(TComponent* Owner);
58:
59:     void SendBCDTo1415(BCD bcd);
60:     bool SetState(int s);
61:     void NextLine();
62:
63:     inline void SetMatrix(int i) { matrix = i; }
64:     void SetMatrix(int x, int y) { matrix = 6*(x-1) + y; }
65:     inline int GetMatrix() { return matrix; }
66:     inline int GetMatrixX() { return (matrix-1)/6 + 1; }
```

```
67:     inline int GetMatrixY() { return (matrix-1)%6 + 1; }
68:     void StepMatrix() { ++matrix; }
69:     void ResetMatrix() { matrix = CONSOLE_MATRIX_HOME; }
70:     void DoMatrix();
71:
72:     void StopPrintOut();
73:     void DoAddressEntry();
74:     void DoDisplay(int phase);
75: };
76: //-----
77: extern TFI1415IO *FI1415IO;
78: //-----
79:
80: //  Keyboard representations of some unusual BCD characters
81:
82: #define KBD_RADICAL      022
83: #define KBD_RECORD_MARK   '|'
84: #define KBD_ALT_BLANK     002
85: #define KBD_WORD_SEPARATOR '^'
86: #define KBD_SEGMENT_MARK  023
87: #define KBD_DELTA          004
88: #define KBD_GROUP_MARK    007
89: #define KBD_WORD_MARK     0x1b
90:
91: #endif
```

```
1: // This Unit defines the behavior of the 1415 console typewriter and
2: // associated things
3:
4: -----
5: #include <vcl\vcl.h>
6: #pragma hdrstop
7:
8: #include "UI1415IO.h"
9: #include "ubcd.h"
10: #include "UI1410DEBUG.h"
11: #include "UI1410CPUT.h"
12:
13: -----
14: #pragma resource "* .dfm"
15: TFI1415IO *FI1415IO;
16: -----
17: __fastcall TFI1415IO::TFI1415IO(TComponent* Owner)
18:     : TForm(Owner)
19: {
20:     state = CONSOLE_IDLE;
21:     Console_AR = 0;
22:     WMCtrlLatch = false;
23:     DisplayWMCtrlLatch = false;
24:     DisplayFullLineLatch = false;
25: }
26: -----
27: void __fastcall TFI1415IO::I1415IOKeyPress(TObject *Sender, char &Key)
28: {
29:     BCD bcd_key;
30:
31:     switch(Key) {
32:     case KBD_RADICAL:
33:         bcd_key = BCD::BCDConvert(ASCII_RADICAL);
34:         break;
35:     case KBD_RECORD_MARK:
36:         bcd_key = BCD::BCDConvert(ASCII_RECORD_MARK);
37:         break;
38:     case KBD_ALT_BLANK:
39:         bcd_key = BCD::BCDConvert(ASCII_ALT_BLANK);
40:         break;
41:     case KBD_WORD_SEPARATOR:
42:         bcd_key = BCD::BCDConvert(ASCII_WORD_SEPARATOR);
43:         break;
44:     case KBD_SEGMENT_MARK:
45:         bcd_key = BCD::BCDConvert(ASCII_SEGMENT_MARK);
46:         break;
47:     case KBD_DELTA:
48:         bcd_key = BCD::BCDConvert(ASCII_DELTA);
49:         break;
50:     case KBD_GROUP_MARK:
51:         bcd_key = BCD::BCDConvert(ASCII_GROUP_MARK);
52:         break;
53:     case KBD_WORD_MARK: // Escape == Wordmark Key
54:         DoWordMark();
55:         return;
56:     case 'b':
57:         bcd_key = BCD::BCDConvert('B');
58:         break;
59:     default:
60:         if(BCD::BCDCheck(Key) < 0) { // Return if unmapped key.
61:             LockKeyboard();
62:             return;
63:         }
64:         bcd_key = BCD::BCDConvert(Key);
65:     }
66: }
```

```
67:     // Decide what to do with key, depending on console state.
68:     // Note that the really special keys (Wordmark, above, and the
69:     // console inquiry buttons), are checked elsewhere. This test
70:     // is for "normal" BCD characters only.
71:
72:     switch(state) {
73:     case CONSOLE_IDLE:
74:         LockKeyboard();                                // Only INQ Request allowed
75:         return;
76:     case CONSOLE_NORMAL:
77:     case CONSOLE_LOAD:
78:     case CONSOLE_ALTER:
79:         UnlockKeyboard();
80:         SendBCDTol415(bcd_key);
81:         break;
82:     case CONSOLE_ADDR:                            // Only digits allowed
83:         if(isdigit(Key)) {
84:             UnlockKeyboard();
85:             SendBCDTol415(bcd_key);
86:
87:             // Ensure that the incoming character has the correct (odd) parity
88:
89:             bcd_key.SetOddParity();
90:
91:             DEBUG("BCD Key set to %d", bcd_key.ToInt())
92:
93:             // During an alteraddress operation, the E Channel (channel 1) gets
94:             // the character on its way to the address register (via the
95:             // A channel and Assembly channel)
96:
97:             CPU -> Channel[CHANNEL1] -> Chr1 -> Set(bcd_key);
98:             *(CPU -> Channel[CHANNEL1] -> Chr2) =
99:                 *(CPU -> Channel[CHANNEL1] -> Chr1);
100:            DoMatrix();
101:        }
102:        else {
103:            LockKeyboard();
104:        }
105:        break;
106:    }
107: }
108: //-----
109: void fastcall TFI1415IO::I1415IOKeyDown(TObject *Sender, WORD &Key,
110:                                         TShiftState Shift)
111: {
112:
113:     // Special key handling: Treat Page Down as a Margin Release.
114:
115:     if(Key == VK_NEXT) {
116:         NextLine();
117:     }
118: }
119: //-----
120:
121: // Method to set the state of the console
122:
123: bool TFI1415IO::SetState(int s)
124: {
125:     switch(s) {
126:     case CONSOLE_IDLE:
127:         InqReq -> Enabled = true;
128:         InqRlse -> Enabled = false;
129:         InqCancel -> Enabled = false;
130:         WordMark -> Enabled = false;
131:         break;
132:     case CONSOLE_NORMAL:
```

```
133:         InqReq -> Enabled = true;
134:         InqRlse -> Enabled = true;
135:         InqCancel -> Enabled = true;
136:         WordMark -> Enabled = false;
137:         break;
138:     case CONSOLE_LOAD:
139:         InqReq -> Enabled = true;
140:         InqRlse -> Enabled = true;
141:         InqCancel -> Enabled = true;
142:         WordMark -> Enabled = true;
143:         break;
144:     case CONSOLE_ALTER:
145:         InqReq -> Enabled = false;
146:         InqRlse -> Enabled = false;
147:         InqCancel -> Enabled = false;
148:         WordMark -> Enabled = true;
149:         break;
150:     case CONSOLE_ADDR:
151:         InqReq -> Enabled = false;
152:         InqRlse -> Enabled = false;
153:         InqCancel -> Enabled = false;
154:         WordMark -> Enabled = false;
155:         switch(CPU -> AddressEntry) {
156:             case CPU -> ADDR_ENTRY_I:
157:                 Console_AR = CPU -> I_AR;
158:                 break;
159:             case CPU -> ADDR_ENTRY_A:
160:                 Console_AR = CPU -> A_AR;
161:                 break;
162:             case CPU -> ADDR_ENTRY_B:
163:                 Console_AR = CPU -> B_AR;
164:                 break;
165:             case CPU -> ADDR_ENTRY_C:
166:                 Console_AR = CPU -> C_AR;
167:                 break;
168:             case CPU -> ADDR_ENTRY_D:
169:                 Console_AR = CPU -> D_AR;
170:                 break;
171:             case CPU -> ADDR_ENTRY_E:
172:                 Console_AR = CPU -> E_AR;
173:                 break;
174:             case CPU -> ADDR_ENTRY_F:
175:                 Console_AR = CPU -> F_AR;
176:                 break;
177:             default:
178:                 DEBUG("1415IO: Invalid CPU Address Entry case:
179: %d",CPU->AddressEntry);
180:                 Console_AR = NULL;
181:             }
182:             if(CPU -> DisplayModeLatch == true) {
183:                 Console_AR = CPU -> C_AR;
184:             }
185:             break;
186:         case CONSOLE_DISPLAY:
187:             InqReq -> Enabled = false;
188:             InqRlse -> Enabled = false;
189:             InqCancel -> Enabled = false;
190:             WordMark -> Enabled = false;
191:             break;
192:         default:
193:             return(false);
194:         }
195:         state = s;
196:         return(true);
197:     }
```

```
198:
199: //
200: // Utility routine to go to next line on console
201: //
202:
203: void TFI1415IO::NextLine()
204: {
205:     I1415IO -> Lines -> Add("");      // Placeholder for wordmarks
206:     I1415IO -> Lines -> Add("");
207: }
208:
209: //
210: // Utility routines to lock/unlock keyboard - display/clear light.
211: //
212:
213: void TFI1415IO::LockKeyboard()
214: {
215:     KeyboardLock -> Enabled = true;
216:     KeyboardLockReset -> Enabled = true;
217: }
218:
219: void TFI1415IO::UnlockKeyboard()
220: {
221:     KeyboardLock -> Enabled = false;
222:     KeyboardLockReset -> Enabled = false;
223: }
224:
225: //
226: // Utility routine to handle wordmark key
227: //
228:
229: void TFI1415IO::DoWordMark()
230: {
231:     int last;
232:
233:     if(state != CONSOLE_LOAD && state != CONSOLE_ALTER) {
234:         LockKeyboard();
235:         return;
236:     }
237:
238:     last = I1415IO -> Lines -> Count - 1;
239:     if(last <= 0) {
240:         NextLine();
241:         last = I1415IO -> Lines -> Count -1;
242:     }
243:
244:     if(I1415IO -> Lines -> Strings[last-1].Length() <=
245:         I1415IO -> Lines -> Strings[last].Length() ) {
246:         I1415IO -> Lines -> Strings[last-1] =
247:             I1415IO -> Lines -> Strings[last-1] + "v";
248:         UnlockKeyboard();
249:     }
250:     else {
251:         LockKeyboard();
252:     }
253: }
254:
255: //
256: // Utility routine to send data to console from
257: //
258:
259: void TFI1415IO::SendBCDTo1415(BCD bcd)
260: {
261:     int last;
262:     char c;
263:
```

```
264:     // Advance to next line if this is the very first line, in
265:     // order to make room for first line of wordmarks
266:
267:     last = I1415IO -> Lines -> Count - 1;
268:     if(last <= 0) {
269:         NextLine();
270:         last = I1415IO -> Lines -> Count -1;
271:     }
272:
273:     // Advance wordmark line to current position if necessary
274:
275:     WMCtrlLatch = bcd.TestWM();
276:     if(I1415IO -> Lines -> Strings[last-1].Length() <=
277:         I1415IO -> Lines -> Strings[last].Length()) {
278:         I1415IO -> Lines -> Strings[last-1] =
279:             I1415IO -> Lines -> Strings[last-1] +
280:                 (WMCtrlLatch ? "v" : " ");
281:     }
282:
283:     // If in display mode (registers or memory), print space as 'b'
284:     // Otherwise, just put the character into the line. (Using ToAscii
285:     // helps us to be invariant about WordMarks and Parity in this
286:     // situation)
287:
288:     c = (state == CONSOLE_DISPLAY && bcd.ToAscii() == ' ') ?
289:         'b' : bcd.ToAscii();
290:
291:     I1415IO -> Lines -> Strings[last] =
292:         I1415IO -> Lines -> Strings[last] + c;
293:
294:     // Advance to next line if this line is full
295:
296:     if(I1415IO -> Lines -> Strings[last].Length() > 79) {
297:         NextLine();
298:         if(state == CONSOLE_DISPLAY) {
299:             DisplayFullLineLatch = true;
300:         }
301:
302:     }
303:
304:     // Tell the form object it has been modified.
305:
306:     I1415IO -> Modified = true;
307:
308:     // Give Windoze a chance to breath
309:
310:     Application -> ProcessMessages();
311:
312: }
313:
314: void TFI1415IO::DoMatrix()
315: {
316:     int savestate;
317:     BCD bcd_char;
318:
319:     if(matrix < 0 || matrix > 42) {
320:         DEBUG("Invalid Console Matrix Position: %d",matrix);
321:         return;
322:     }
323:
324:     DEBUG("Doing Console Matrix Position %d",matrix);
325:
326:     // Console Matrix Processing
327:
328:     // Positions 0 (Home) thru 36 are used for display/output purposes
329:
```

```

330:     switch(matrix) {
331:
332:     case 0:
333:         /* Home position: do nothing
334:          SetState(CONSOLE_IDLE);
335:          break;
336:
337:     case 1:
338:     case 2:
339:     case 3:
340:     case 4:
341:     case 5:
342:         SendBCDTo1415(CPU -> I_AR -> GateBCD(matrix));
343:         break;
344:
345:     case 6:
346:     case 12:
347:     case 18:
348:     case 21:
349:     case 25:
350:     case 31:
351:     case 36:
352:         savestate = state;
353:         state = CONSOLE_NORMAL;           // So space prints as one !!
354:         SendBCDTo1415(BCD::BCDConvert(' '));
355:         state = savestate;
356:         break;
357:
358:     case 7:
359:     case 8:
360:     case 9:
361:     case 10:
362:     case 11:
363:         SendBCDTo1415(CPU -> A_AR -> GateBCD(matrix-6));
364:         break;
365:
366:     case 13:
367:     case 14:
368:     case 15:
369:     case 16:
370:     case 17:
371:         SendBCDTo1415(CPU -> B_AR -> GateBCD(matrix-12));
372:         break;
373:
374:     case 19:
375:         SendBCDTo1415(CPU -> Op_Reg -> Get());
376:         break;
377:
378:     case 20:
379:         SendBCDTo1415(CPU -> Op_Mod_Reg -> Get());
380:         break;
381:
382:     case 22:
383:         SendBCDTo1415(CPU -> A_Reg -> Get());
384:         break;
385:
386:     case 23:
387:         SendBCDTo1415(CPU -> B_Reg -> Get());
388:         break;
389:
390:     case 24:
391:         SendBCDTo1415(CPU -> AssemblyChannel());
392:         break;
393:
394:     case 26:
395:         SendBCDTo1415(CPU -> Channel[CHANNEL1] -> ChUnitType -> Get());

```

Select()

```

396:         break;
397:
398:     case 27:
399:         SendBCDTo1415(CPU -> Channel[CHANNEL1] -> ChUnitNumber -> Get());
400:         break;
401:
402:     case 28:
403: #if MAXCHANNEL > 1
404:         SendBCDTo1415(CPU -> Channel[CHANNEL2] -> ChUnitType -> Get());
405: #endif
406:         break;
407:
408:     case 29:
409: #if MAXCHANNEL > 1
410:         SendBCDTo1415(CPU -> Channel[CHANNEL2] -> ChUnitNumber -> Get());
411: #endif
412:         break;
413:
414:     case 30:
415:         if(CPU -> DisplayModeLatch) {
416:             SendBCDTo1415(BCD::BCDConvert('D'));
417:         }
418:         break;
419:
420:     case 32:
421:         if(CPU -> DisplayModeLatch) {
422:             SendBCDTo1415(CPU -> B_Reg -> Get());
423:         }
424:         StepMatrix();
425:         break;
426:
427:     case 33:
428:         if(CPU -> DisplayModeLatch) {
429:             SendBCDTo1415(CPU -> B_Reg -> Get());
430:         }
431:         DisplayWMCtrlLatch = WMCtrlLatch;
432:         break;
433:
434:     case 34:
435:         NextLine();
436:         break;
437:
438: // Positions 37 thru 42 are to accept an address for address set.
439: // For 41, do the last digit, then fall thru to processing for 42.
440:
441:     case 37:
442:     case 38:
443:     case 39:
444:     case 40:
445:     case 41:
446:         bcd_char = CPU -> Channel[CHANNEL1] -> ChR2 -> Get();
447:         DEBUG("Value to set into register from console is %d", bcd_char.ToInt());
448:         Console_AR -> Set(TWOOF5(bcd_char), matrix-36);
449:         StepMatrix();
450:         if(matrix != 42) {
451:             break;
452:         }
453:
454: // Fall thru to next entry for last digit!!
455:
456: // At the end of an address set, go to next line, and reset console
457:
458:     case 42:
459:         NextLine();
460:         if(CPU -> DisplayModeLatch) {
461:             SetState(CONSOLE_DISPLAY);

```

much changed

Much change!

channel (via A channel)

```
462:             DoDisplay(2);
463:         }
464:     else {
465:         ResetMatrix();
466:         SetState(CONSOLE_IDLE);
467:     }
468:     break;
469:
470:     default:
471:         break;
472:     }
473: }
474:
475: // Handle a console stop print-out operation
476:
477: void TFI1415IO::StopPrintOut()
478: {
479:     SetState(CONSOLE_DISPLAY);
480:     NextLine();
481:     SetMatrix(35);
482:     SendBCDTo1415(BCD::BCDConvert('S'));
483:     StepMatrix();
484:     DoMatrix();
485:     SetMatrix(1);
486:     while(GetMatrixX() < 5) {
487:         DoMatrix();
488:         StepMatrix();
489:     }
490:
491:
492:     while(GetMatrixY() < 6) {
493:         DoMatrix();
494:         StepMatrix();
495:     }
496:     SetMatrix(34);
497:     DoMatrix();
498:     SetMatrix(CONSOLE_MATRIX_HOME);
499:     DoMatrix();
500:     CPU -> Display();
501: }
502:
503: // Initiate a Console Address Set operation
504:
505: void TFI1415IO::DoAddressEntry()
506: {
507:     SetState(CONSOLE_ADDR);
508:     Console_AR -> Reset();
509:     SetMatrix(35);
510:     if(CPU -> DisplayModeLatch) {
511:         SendBCDTo1415(BCD::BCDConvert('D'));
512:     }
513:     else {
514:         SendBCDTo1415(BCD::BCDConvert((Console_AR == CPU -> I_AR) ? 'B' : '#'));
515:     }
516:     StepMatrix();
517:     DoMatrix();
518:     SetMatrix(37);
519:
520:     // The console matrix code will automatically handle the rest, so
521:     // just return!
522:
523: }
524:
525: // Initiate a Memory Display operation
526:
527: void TFI1415IO::DoDisplay(int phase)
```

```
528: {
529:     // Phase 1 takes us thru the address entry. When the address entry
530:     // is done, the code in the matrix notices we are doing a display,
531:     // and fires up Phase 2 (like co-routines)
532:
533:     if(phase == 1) {
534:         if(DisplayWMCtrlLatch) {
535:             DisplayWMCtrlLatch = false;
536:             phase = 3;
537:         }
538:     else {
539:         CPU -> StopKeyLatch = false;
540:         CPU -> IRing -> Reset();
541:         WMCtrlLatch = DisplayWMCtrlLatch = DisplayFullLineLatch = false;
542:
543:         // The DisplayModeLatch causes No A Ch, B Ch, Address Wrap Checks
544:         // It also forces the Address Entry routine to print 'D', and put
545:         // the resultant address into CAR
546:
547:         CPU -> DisplayModeLatch = true;
548:
549:         DoAddressEntry();
550:         return;
551:     }
552: }
553:
554: // Phase 2 handles the beginning of the actual display part
555:
556: if(phase == 2) {
557:     matrix = 30;
558:     DoMatrix();
559:     StepMatrix();
560:     DoMatrix();
561:     CPU -> SetScan(SCAN_2);
562:     CPU -> CycleRing -> Set(CYCLE_D);
563:     CPU -> STAR = CPU -> C_AR;
564:     CPU -> Readout();
565:     StepMatrix();
566:     phase = 3;
567: }
568:
569: if(phase == 3) {
570:     while(true) {
571:         if(DisplayWMCtrlLatch) {
572:             break;
573:         }
574:         DoMatrix();
575:         CPU -> D_AR -> Set(CPU -> STARScan());
576:         CPU -> SetScan(SCAN_2);
577:         CPU -> CycleRing -> Set(CYCLE_D);
578:         CPU -> STAR = CPU -> D_AR;
579:         if(CPU -> StopKeyLatch) {
580:             phase = 4;
581:             break;
582:         }
583:         else {
584:             CPU -> Readout();
585:         }
586:     }
587: }
588:
589: if(phase == 4 && state == CONSOLE_DISPLAY) {
590:     DisplayWMCtrlLatch = false;
591:     StepMatrix();
592:     DoMatrix();
593:     ResetMatrix();
```

```
594:         SetState(CONSOLE_IDLE);
595:     }
596: }
597:
598: void __fastcall TFI1415IO::WordMarkClick(TObject *Sender)
599: {
600:     DoWordMark();
601:     FocusControl(I1415IO);
602: }
603: //-----
604: void __fastcall TFI1415IO::MarginReleaseClick(TObject *Sender)
605: {
606:     NextLine();
607:     FocusControl(I1415IO);
608:     if(state == CONSOLE_DISPLAY) {
609:         DisplayFullLineLatch = true;
610:     }
611: }
612: //-----
613: void __fastcall TFI1415IO::KeyboardLockResetTimer(TObject *Sender)
614: {
615:     UnlockKeyboard();
616: }
617: //-----
```

```

1: //-----
2: #ifndef UI1415CEH
3: #define UI1415CEH
4: //-----
5: #include <vc1\Classes.hpp>
6: #include <vc1\Controls.hpp>
7: #include <vc1\StdCtrls.hpp>
8: #include <vc1\Forms.hpp>
9: //-----
10: class TFI1415CE : public TForm
11: {
12:     __published: // IDE-managed Components
13:     TComboBox *AddressEntry;
14:     TLabel *Label1;
15:     TComboBox *StorageScan;
16:     TLabel *Label2;
17:     TComboBox *CycleControl;
18:     TLabel *Label3;
19:     TComboBox *CheckControl;
20:     TLabel *Label4;
21:     TCheckBox *DiskWrInhibit;
22:     TComboBox *DensityCh1;
23:     TLabel *Label5;
24:     TComboBox *DensityCh2;
25:     TLabel *Label6;
26:     TButton *StartPrintOut;
27:     TCheckBox *Compat1401;
28:     TButton *CheckReset1401;
29:     TCheckBox *CheckStop1401;
30:     TButton *CheckTest1;
31:     TButton *CheckTest2;
32:     TButton *CheckTest3;
33:     TLabel *Label7;
34:     TLabel *Label8;
35:     TCheckBox *AsteriskInsert;
36:     TCheckBox *InhibitPrintOut;
37:     TCheckBox *BitSenseC;
38:     TCheckBox *BitSenseB;
39:     TCheckBox *BitSenseA;
40:     TCheckBox *BitSense8;
41:     TCheckBox *BitSense4;
42:     TCheckBox *BitSense2;
43:     TCheckBox *BitSense1;
44:     TCheckBox *BitSenseWM;
45:     TLabel *Label9;
46:     TLabel *Label10;
47:     TLabel *Label11;
48:     void __fastcall AddressEntryChange(TObject *Sender);
49:     void __fastcall StorageScanChange(TObject *Sender);
50:     void __fastcall CycleControlChange(TObject *Sender);
51:     void __fastcall CheckControlChange(TObject *Sender);
52:     void __fastcall DiskWrInhibitClick(TObject *Sender);
53:     void __fastcall DensityCh1Change(TObject *Sender);
54:     void __fastcall DensityCh2Change(TObject *Sender);
55:     void __fastcall AsteriskInsertClick(TObject *Sender);
56:     void __fastcall InhibitPrintOutClick(TObject *Sender);
57:     void __fastcall BitSenseCClick(TObject *Sender);
58:     void __fastcall BitSenseBClick(TObject *Sender);
59:     void __fastcall BitSenseAClick(TObject *Sender);
60:     void __fastcall BitSense8Click(TObject *Sender);
61:     void __fastcall BitSense4Click(TObject *Sender);
62:     void __fastcall BitSense2Click(TObject *Sender);
63:     void __fastcall BitSense1Click(TObject *Sender);
64:     void __fastcall BitSenseWMClick(TObject *Sender);
65: 
```

*#include "UI1415FO.h"*

*StartPrintOutClick()*

```
67: private:    // User declarations
68:     int SenseBit;
69:     int SetSense(bool b,int i);
70: public:     // User declarations
71:     __fastcall TFI1415CE(TComponent* Owner);
72: };
73: //-----
74: extern TFI1415CE *FI1415CE;
75: //-----
76: #endif
```

```
1: //-----
2: #include <vcl\vcl.h>
3: #pragma hdrstop
4:
5: #include "UI1415CE.h"
6: //-----
7: #pragma resource "* .dfm"
8:
9: #include "UI1410CPU.T"
10: #include "UI1410DEBUG.h"
11:
12: TFI1415CE *FI1415CE;
13: //-----
14: __fastcall TFI1415CE::TFI1415CE(TComponent* Owner)
15:     : TForm(Owner)
16: {
17:     WindowState = wsMinimized;
18:     AddressEntry -> ItemIndex = CPU -> ADDR_ENTRY_I;
19:     StorageScan -> ItemIndex = CPU -> SSCAN_OFF;
20:     CycleControl -> ItemIndex = CPU -> CYCLE_OFF;
21:     CheckControl -> ItemIndex = CPU -> CHECK_STOP;
22:     DiskWrInhibit -> Checked = false;
23:     DensityCh1 -> ItemIndex = 0;
24:     DensityCh2 -> ItemIndex = 0;
25:     AsteriskInsert -> Checked = true;
26:     InhibitPrintOut -> Checked = false;
27:     SenseBit = 0;
28:     BitSenseC -> Checked = false;
29:     BitSenseB -> Checked = false;
30:     BitSenseA -> Checked = false;
31:     BitSense8 -> Checked = false;
32:     BitSense4 -> Checked = false;
33:     BitSense2 -> Checked = false;
34:     BitSense1 -> Checked = false;
35:     BitSenseWM -> Checked = false;
36: }
37: //-----
38: void __fastcall TFI1415CE::AddressEntryChange(TObject *Sender)
39: {
40:     if(AddressEntry -> ItemIndex >= 0) {
41:         CPU -> AddressEntry = AddressEntry -> ItemIndex;
42:     }
43:     CPU -> OffNormal -> Display();
44:     DEBUG("Address Entry Switch set to %d",CPU -> AddressEntry)
45: }
46: //-----
47: void __fastcall TFI1415CE::StorageScanChange(TObject *Sender)
48: {
49:     if(StorageScan -> ItemIndex >= 0) {
50:         CPU -> StorageScan = StorageScan -> ItemIndex;
51:     }
52:     CPU -> OffNormal -> Display();
53:     DEBUG("Storage Scan Switch set to %d",CPU -> StorageScan)
54: }
55: //-----
56: void __fastcall TFI1415CE::CycleControlChange(TObject *Sender)
57: {
58:     if(CycleControl -> ItemIndex >= 0) {
59:         CPU -> CycleControl = CycleControl -> ItemIndex;
60:     }
61:     CPU -> OffNormal -> Display();
62:     DEBUG("Cycle Control Switch set to %d",CPU -> CycleControl)
63: }
64: //-----
65: void __fastcall TFI1415CE::CheckControlChange(TObject *Sender)
66: {
```

```
67:     if(CheckControl -> ItemIndex >= 0) {
68:         CPU -> CheckControl = CheckControl -> ItemIndex;
69:     }
70:     CPU -> OffNormal -> Display();
71:     DEBUG("Check Control Switch set to %d",CPU -> CheckControl)
72: }
73: //-----
74: void __fastcall TFI1415CE::DiskWrInhibitClick(TObject *Sender)
75: {
76:     CPU -> DiskWrInhibit = DiskWrInhibit -> Checked;
77:     DEBUG("Disk Write Inhibit Switch set to %d",CPU -> DiskWrInhibit)
78: }
79: //-----
80: void __fastcall TFI1415CE::DensityCh1Change(TObject *Sender)
81: {
82:     if(DensityCh1 -> ItemIndex >= 0) {
83:         CPU -> Channel[CHANNEL1] -> TapeDensity = DensityCh1 -> ItemIndex;
84:     }
85:     DEBUG("Channel 1 Tape Density Switch set to %d",
86:           CPU -> Channel[CHANNEL1] -> TapeDensity)
87: }
88: //-----
89: void __fastcall TFI1415CE::DensityCh2Change(TObject *Sender)
90: {
91:     if(DensityCh2 -> ItemIndex >= 0) {
92:         CPU -> Channel[CHANNEL2] -> TapeDensity = DensityCh2 -> ItemIndex;
93:     }
94:     DEBUG("Channel 2 Tape Density Switch set to %d",
95:           CPU -> Channel[CHANNEL2] -> TapeDensity)
96: }
97: //-----
98: void __fastcall TFI1415CE::AsteriskInsertClick(TObject *Sender)
99: {
100:    CPU -> AsteriskInsert = AsteriskInsert -> Checked;
101:    CPU -> OffNormal -> Display();
102:    DEBUG("Asterisk Insert Switch set to %d",CPU -> AsteriskInsert)
103: }
104: //-----
105: void __fastcall TFI1415CE::InhibitPrintOutClick(TObject *Sender)
106: {
107:    CPU -> InhibitPrintOut = InhibitPrintOut -> Checked;
108:    CPU -> OffNormal -> Display();
109:    DEBUG("Inhibit Print Out Switch set to %d",CPU -> InhibitPrintOut)
110: }
111: //-----
112: void __fastcall TFI1415CE::BitSenseCClick(TObject *Sender)
113: {
114:     SetSense(BitSenseC -> Checked,BITC);
115: }
116: //-----
117:
118: int TFI1415CE::SetSense(bool b,int i)
119: {
120:     SenseBit = (b ? (SenseBit | i) : (SenseBit & ~i));
121:     CPU -> BitSwitches = BCD(SenseBit);
122:     DEBUG("Bit/Sense Switches now set to 0x%x",CPU -> BitSwitches.ToInt())
123: }
124: void __fastcall TFI1415CE::BitSenseBClick(TObject *Sender)
125: {
126:     SetSense(BitSenseB -> Checked,BITB);
127: }
128: //-----
129: void __fastcall TFI1415CE::BitSenseAClick(TObject *Sender)
130: {
131:     SetSense(BitSenseA -> Checked,BITA);
132: }
```

```
133: //-----
134: void __fastcall TFI1415CE::BitSense8Click(TObject *Sender)
135: {
136:     SetSense(BitSense8 -> Checked,BIT8);
137: }
138: //-----
139: void __fastcall TFI1415CE::BitSense4Click(TObject *Sender)
140: {
141:     SetSense(BitSense4 -> Checked,BIT4);
142: }
143: //-----
144: void __fastcall TFI1415CE::BitSense2Click(TObject *Sender)
145: {
146:     SetSense(BitSense2 -> Checked,BIT2);
147: }
148: //-----
149: void __fastcall TFI1415CE::BitSense1Click(TObject *Sender)
150: {
151:     SetSense(BitSense1 -> Checked,BIT1);
152: }
153: //-----
154: void __fastcall TFI1415CE::BitSenseWMClick(TObject *Sender)
155: {
156:     SetSense(BitSenseWM -> Checked,BITWM);
157: }
158: //-----
```

StartPrintOutClick ()

```
1: //-----  
2: #ifndef UI1410PWRH  
3: #define UI1410PWRH  
4: //-----  
5: #include <vcl\Classes.hpp>  
6: #include <vcl\Controls.hpp>  
7: #include <vcl\StdCtrls.hpp>  
8: #include <vcl\Forms.hpp>  
9: #include <vcl\ExtCtrls.hpp>  
10: #include <vcl\Buttons.hpp>  
11: //-----  
12:  
13: #include "UI1410CPUT.H"  
14: #include "UI14101.H"      #include "UI141010.H"  
15:  
16: // 1410 Power panel  
17:  
18: class TFI1410PWR : public TForm  
19: {  
20:     __published: // IDE-managed Components  
21:         TPanel *Panel1;  
22:         TLabel *EMERGENCY;  
23:         TLabel *OFF;  
24:         TBitBtn *EmergencyOff;  
25:         TBitBtn *ComputerReset;  
26:         TBitBtn *DCOff;  
27:         TPanel *READY;  
28:         TBitBtn *PowerOff;  
29:         TPanel *Panel2;  
30:         TComboBox *Mode;  
31:         TLabel *MODELABEL;  
32:         TBitBtn *Start;  
33:         TBitBtn *Stop;  
34:         TBitBtn *ProgramReset;  
35:         void __fastcall EmergencyOffClick(TObject *Sender);  
36:         void __fastcall ComputerResetClick(TObject *Sender);  
37:         void __fastcall ModeChange(TObject *Sender);  
38:         void __fastcall ProgramResetClick(TObject *Sender);  
39:     private: // User declarations  
40:     public: // User declarations  
41:         __fastcall TFI1410PWR(TComponent* Owner);  
42: };  
43: //-----  
44: extern TFI1410PWR *FI1410PWR;  
45: //-----  
46: #endif
```

StartClick( )  
StopClick( )

```
1: //-----
2: #include <vcl\vcl.h>
3: #pragma hdrstop
4:
5: #include "UI1410PWR.h"
6: //-----
7: #pragma resource "* .dfm"
8:
9: #include "UI1410DEBUG.h"
10:
11: // These methods define the operation of the 1410 power panel
12:
13: TFI1410PWR *FI1410PWR;
14: //-----
15: __fastcall TFI1410PWR::TFI1410PWR(TComponent* Owner)
16:   : TForm(Owner)
17: {
18:   Mode -> ItemIndex = CPU -> MODE_RUN;
19:   Height = 522;
20:   Width = 327;
21: }
22: //-----
23: void __fastcall TFI1410PWR::EmergencyOffClick(TObject *Sender)
24: {
25:   FI14101 -> Close();
26: }
27: //-----
28: void __fastcall TFI1410PWR::ComputerResetClick(TObject *Sender)
29: {
30:   TCpuObject *o;
31:
32:   DEBUG("Computer Reset",0)
33:   for(o = CPU -> ResetList; o != 0; o = o -> NextReset) {
34:     o -> OnComputerReset();
35:   }
36:
37:   // Handle any special case latches (which are reset in the above loop)
38:
39:   CPU -> CompareBLTA -> Set();
40:   CPU -> I_AR -> Set(1);
41:
42:   // Hitting Computer Reset is like clicking STOP too
43:
44:   StopClick(Sender);
45:
46:   // Reset any simple latches as needed
47:
48:   CPU -> StopLatch = true;
49:   CPU -> DisplayModeLatch = false;
50:   CPU -> StorageWrapLatch = false;
51:   CPU -> ProcessRoutineLatch = false;
52:   CPU -> BranchToILatch = true;
53:   CPU -> BranchLatch = true;
54:   CPU -> IRingControl = true;
55:   CPU -> IndexLatches = 0;
56:
57:   // Reset the console matrix
58:
59:   FI1415IO -> ResetMatrix();
60:
61:   // Do a display
62:
63:   CPU -> Display();
64:
65:   // Reset the console
66:
```

```
67:     FI1415IO -> NextLine();
68:     FI1415IO -> SetState(CONSOLE_IDLE);
69: }
70: //-----
71: void __fastcall TFI1410PWR::ModeChange(TObject *Sender)
72: {
73:     if(Mode -> ItemIndex >= 0) {
74:         CPU -> Mode = Mode -> ItemIndex;
75:     }
76:
77:     // Moving the mode switch is equivalent to hitting STOP
78:
79:     StopClick(Sender);
80:
81:     CPU -> OffNormal -> Display();
82:     DEBUG("Mode Switch set to %d",CPU -> Mode)
83: }
84: //-----
85: void __fastcall TFI1410PWR::ProgramResetClick(TObject *Sender)
86: {
87:     TCpuObject *o;
88:
89:     DEBUG("Program Reset",0)
90:     for(o = CPU -> ResetList; o != 0; o = o -> NextReset) {
91:         o -> OnProgramReset();
92:     }
93:
94:     // Handle any special cases
95:
96:     CPU -> StopLatch = true;
97:     CPU -> BranchToILatch = true;           // Causes I Fetch to start at 00001
98:     CPU -> BranchLatch = true;
99:     CPU -> IndexLatches = 0;
100:
101:    StopClick(Sender);
102:
103:    CPU -> Display();
104: }
105: //-----
106:
107: void __fastcall TFI1410PWR::StartClick(TObject *Sender)
108: {
109:     switch(CPU -> Mode) {
110:
111:     case CPU -> MODE_ADDR:
112:         CPU -> ProcessRoutineLatch = false;
113:         FI1415IO -> DoAddressEntry();
114:         CPU -> BranchToILatch = false;
115:         CPU -> BranchLatch = false;
116:         break;
117:
118:     case CPU -> MODE_DISPLAY:
119:         CPU -> ProcessRoutineLatch = false;
120:         FI1415IO -> DoDisplay(1);
121:         break;
122:
123:     case CPU -> MODE_ALTER:
124:         CPU -> ProcessRoutineLatch = false;
125:         FI1415IO -> DoAlter();
126:         break;
127:
128:     case CPU -> MODE_IE:
129:     case CPU -> MODE_RUN:
130:
131:         // We loop here until something makes us stop and return,
132:         // such as a special cycle control setting, I/E mode when
```

```
133:         // finished fetching, the STOP key at the end of an instruction,
134:         // etc.
135:
136:         // NOTE: This code will almost certainly get moved to UI1410INST
137:         // someday/
138:
139:     CPU -> StopLatch = false;
140:
141:     while(true) {
142:
143:         // I Cycle start
144:
145:         CPU -> ProcessRoutineLatch = true;
146:
147:         if(CPU -> IRingControl) {
148:             CPU -> InstructionDecodeStart();
149:             if(CPU -> StopLatch) {
150:                 FI1415IO -> StopPrintOut('S');
151:                 CPU -> Display();
152:                 return;
153:             }
154:             if(CPU -> CycleControl != CPU -> CYCLE_OFF) {
155:                 FI1415IO -> StopPrintOut('C');
156:                 CPU -> Display();
157:                 break;
158:             }
159:         }
160:
161:         // I Cycle
162:
163:         else if(!CPU -> LastInstructionReadout &&
164:                 CPU -> CycleRing -> State() == CYCLE_I) {
165:             CPU -> InstructionDecode();
166:             if(CPU -> StopLatch) {
167:                 FI1415IO -> StopPrintOut('S');
168:                 return;
169:             }
170:             if(CPU -> CycleControl != CPU -> CYCLE_OFF) {
171:                 FI1415IO -> StopPrintOut('C');
172:                 CPU -> Display();
173:             }
174:         }
175:
176:         // X (index) Cycle
177:
178:         else if(!CPU -> LastInstructionReadout &&
179:                 CPU -> CycleRing -> State() == CYCLE_X) {
180:             CPU -> InstructionIndex();
181:             if(CPU -> StopLatch) {
182:                 FI1415IO -> StopPrintOut('S');
183:                 return;
184:             }
185:             if(CPU -> CycleControl != CPU -> CYCLE_OFF) {
186:                 FI1415IO -> StopPrintOut('C');
187:                 CPU -> Display();
188:             }
189:         }
190:
191:         // Execute Cycle would go here....
192:
193:         if(CPU -> LastInstructionReadout) {
194:
195:             // Temporarily use LastInstructionReadout to set IRingControl
196:             // In otherwords, all instructions do nothing.
197:
198:             CPU -> IRingControl = true;
```

```
199:             CPU -> LastInstructionReadout = false;
200:
201:             // If in IE cycle, print out after instruction fetch
202:             // then break out. (Without the preceeding, we would
203:             // then come back and execute the instruction.
204:
205:             if(CPU -> Mode == CPU -> MODE_IE) {
206:                 FI1415IO -> StopPrintOut('C');
207:                 break;
208:             }
209:         }
210:
211:         // If we are in storage/logic cycle, break out.
212:
213:         if(CPU -> CycleControl != CPU -> CYCLE_OFF) {
214:             break;
215:         }
216:
217:         // If we aren't in Storage Cycle or I/E Cycle Mode...
218:
219:         // Give Windoze a chance to breathe
220:
221:         Application -> ProcessMessages();
222:     }
223:
224:     break;
225:
226:     default:
227:         break;
228:     }
229:
230: }
231: //-----
232: void __fastcall TFI1410PWR::StopClick(TObject *Sender)
233: {
234:     CPU -> StopKeyLatch = true;
235:     CPU -> ProcessRoutineLatch = false;
236:
237:     FI1415IO -> DoDisplay(4);
238: }
239: //-----
```



```
1: //-----  
2: #ifndef UI1410INSTH  
3: #define UI1410INSTH  
4: //-----  
5:  
6: #include "ubcd.h"  
7: #include "UI1410CPUT.H"  
8: #include "UI1410DEBUG.H"  
9:  
10: #include <assert.h>  
11:  
12: // Op code table Instruction Readout lines values: OpReadoutLines  
13:  
14: #define OP_PERCENTTYPE 1  
15: #define OP_NOTPERCENTTYPE 2  
16: #define OP_ADDRDBL 4  
17: #define OP_NOTADDRDBL 8  
18: #define OP_1ADDRPLUSMOD 16  
19: #define OP_2ADDRNOMOD 32  
20: #define OP_2ADDRPLUSMOD 64  
21: #define OP_2ADDRESS 128  
22: #define OP_ADDRTYPE 256  
23: #define OP_2CHARONLY 512  
24: #define OP_CCYCLE 1024  
25: #define OP_NOCORDCY 2048  
26: #define OP_NODCYIRING6 4096  
27: #define OP_NOINDEXON1STADDR 8192  
28:  
29: // Op code table Operational lines values: OpOperationalLines  
30:  
31: #define OP_RESETTYPE 1  
32: #define OP_ADDORSUBT 2  
33: #define OP_MPYORDIV 4  
34: #define OP_ADDTYPE 8  
35: #define OP_ARITHTYPE 16  
36: #define OP_EORZ 32  
37: #define OP_COMPARETYPE 64  
38: #define OP_BRANCHTYPE 128  
39: #define OP_NOBRANCH 256  
40: #define OP_WORDMARK 512  
41: #define OP_MORL 1024  
42:  
43: // Op code table Control lines values: OpControlLines  
44:  
45: #define OP_1STSCANFIRST 1  
46: #define OP_ACYFIRST 2  
47: #define OP_STDACYCLE 4  
48: #define OP_BCYFIRST 8  
49: #define OP_AREGTOACHONBCY 16  
50: #define OP_OPMODTOACHONBCY 32  
51: #define OP_LOADMEMONBCY 64  
52: #define OP_RGENMEMONBCY 128  
53: #define OP_STOPATFONBCY 256  
54: #define OP_STOPATHONBCY 512  
55: #define OP_STOPATJONBCY 1024  
56: #define OP_ROBARONSCANBCY 2048  
57: #define OP_ROAARONACY 4096  
58:  
59: #define OP_INVALID 65535  
60:  
61: #define OP_NOP 37  
62: #define OP_TABLESEARCH 19  
63: #define OP_SAR_G 55  
64:  
65: #endif
```

```
1: //-----
2: #include <vcl\vcl.h>
3: #pragma hdrstop
4:
5: #include "UI1410INST.H"
6: //-----
7:
8: // This module handles Instruction Decode and Execution in the CPU
9:
10: /* The following table is given in the order of the 1410 BCD codes, and
11:    contains the opcode common lines - 3 16 bit words.
12: */
13:
14: struct OpCodeCommonLines OpCodeTable[64] = {
15:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 00 spc */
16:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 01 1 */
17:     { 8+512, 256, 32+128 }, /* 02 2 */
18:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 03 3 */
19:     { 8+512, 256, 32+128 }, /* 04 4 */
20:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 05 5 */
21:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 06 6 */
22:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 07 7 */
23:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 08 8 */
24:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 09 9 */
25:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 10 0 */
26:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 11 = */
27:     { 2+8+32+128+256+1024, 4+16+256, 1+2+4+16+64+256+1024 }, /* 12 @ */
28:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 13 : */
29:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 14 > */
30:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 15 rad */
31:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 15 alt */
32:     { 2+4+32+128+256+2048+4096, 128, 1+8+16+256+2048 }, /* 17 / */
33:     { 2+4+32+128+256+4096, 2+8+16+256, 1+2+4+16+64+4096 }, /* 18 S */
34:     { 2+8+64+128+256+1024+4096, 64+256, 1+2+4+16+128+2048 }, /* 19 T */
35:     { 1+8+16+256+8192, 256, 32 }, /* 20 U */
36:     { 2+8+64+128+256+2048+4096, 128, 1+8+32+128+512+2048 }, /* 21 V */
37:     { 2+8+64+128+256+2048+4096, 128, 1+8+32+128+512+2048 }, /* 22 W */
38:     { 2+4+16+256, 128, 32+128+256+2048 }, /* 23 X */
39:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 24 Y */
40:     { 2+8+32+128+256+2048+4096, 32+256, 1+2+4+16+64+1024+2048+4096 }, /* 25 Z */
41:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 26 RM */
42:     { 2+4+32+128+256+2048+4096, 256+512, 1+2+16+2048+4096 }, /* 27 , */
43:     { 2+8+32+128+256+1024, 4+16+256, 1+2+4+16+64+256+1024 }, /* 28 % */
44:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 29 WS */
45:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 30 \ */
46:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 31 SM */
47:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 32 - */
48:     { 2+4+16+256+2048, 128, 32+128+256+2048 }, /* 33 J */
49:     { 8+512, 256, 32+128 }, /* 34 K */
50:     { 1+8+64+128+256+8192, 256+1024, 0 }, /* 35 L */
51:     { 1+8+64+128+256+8192, 256+1024, 0 }, /* 36 M */
52:     { 0, 0, 0 }, /* 37 N */
53:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 38 O */
54:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 39 P */
55:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 40 Q */
56:     { 2+4+16+256, 128, 32+128+256+2048 }, /* 41 R */
57:     { 2+4+32+128+256+4096, 1+8+16+256, 1+2+4+16+64+256+1024+4096 }, /* 42 ! */
58:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 43 $ */
59:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 44 * */
60:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 45 ] */
61:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 46 ; */
62:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 47 Delt */
63:     { OP_INVALID, OP_INVALID, OP_INVALID }, /* 48 + */
64:     { 2+4+32+128+256+4096, 2+8+16+256, 1+2+4+16+64+4096 }, /* 49 A */
65:     { 2+8+64+128+256+2048+4096, 64+128, 1+8+32+128+512+2048 }, /* 52 B */
66:     { 2+8+32+128+256+2048+4096, 64+256, 1+2+4+16+128+2048+4096 }, /* 51 C */
```

```
67: { 2+8+64+128+256+2048+4096, 256, 2+4+16+64+256+1024+2048+4096 }, /* 52 D */
68: { 2+8+32+128+256+2048+4096, 32+256, 1+2+4+16+64+1024+2048+4096 }, /* 53 E */
69: { 8+512, 256, 32+128 }, /* 54 F */
70: { 8+16+256+8192, 256, 1 }, /* 55 G */
71: { OP_INVALID, OP_INVALID, OP_INVALID }, /* 56 H */
72: { OP_INVALID, OP_INVALID, OP_INVALID }, /* 57 I */
73: { 2+4+32+128+256+4096, 1+8+16+256, 1+2+4+16+64+256+1024+4096 }, /* 58 ? */
74: { 2+8+256+2048+4096, 128, 16+128+256+2048 }, /* 59 . */
75: { 2+4+32+128+256+2048+4096, 256+512, 1+2+16+2048+4096 }, /* 60 loz.*/
76: { OP_INVALID, OP_INVALID, OP_INVALID }, /* 61 [*]
77: { OP_INVALID, OP_INVALID, OP_INVALID }, /* 62 < */
78: { OP_INVALID, OP_INVALID, OP_INVALID } /* 63 GM */

79: };
80:
81: // Table indicating when zones are valid for ops with addresses
82:
83: static bool IRingZoneTable [] = {
84:     false, false, false, true, true, false, false, true, true,
85:     false, true, true
86: };
87:
88:
89: // Table of Index Register locations
90:
91: int IndexRegisterLookup[16] = {
92:     0,29,34,39,44,49,54,59,64,69,74,79,84,89,94,99
93: };
94:
95:
96: // Instruction Decode - initial phase
97:
98: void T1410CPU::InstructionDecodeStart()
99: {
100:
101:     int op_bin;
102:
103:     SetScan(SCAN_N);           // During I phase, no storage scan mode set
104:     CycleRing -> Set(CYCLE_I); // Doing I cycles
105:     IRing -> Reset();        // Reset I Ring to Op state
106:     LastInstructionReadout = false; // Set true at end of fetch
107:     IRingControl = false;      // Reset I Ring control state
108:
109:     // Figure out where to go. If we are branching, then branch to AAR
110:     // unless the BranchToILatch (e.g. Program Reset) is set. If not
111:     // branching, just use the IAR. STAR is set to the location to begin
112:     // instruction readout.
113:
114:     if(BranchLatch) {
115:         if(BranchToILatch) {
116:             STAR -> Set(1);
117:         }
118:         else {
119:             *STAR = *A_AR;
120:         }
121:     }
122:     else {
123:         *STAR = *I_AR;
124:     }
125:
126:     // Fetch the instruction code. Check to make sure it has it's wordmark
127:     // If not, stop with an instruction check.
128:
129:     // Take an I Cycle
130:
131:     Readout();
132:     Cycle();
```

```
133:     I_AR -> Set(STARMod(+1));           // This one *always* advances I_AR
134:
135:     if(!B_Reg -> Get().TestWM()) {
136:         InstructionCheck -> SetStop("Instruction Check: No WordMark present");
137:         return;
138:     }
139:
140:     // Copy the opcode into the Op register, and decode it.
141:
142:     *Op_Reg = *B_Reg;
143:
144:     op_bin = Op_Reg -> Get().ToInt() & 0x3f;
145:     OpReadOutLines = OpCodeTable[op_bin].ReadOut;
146:     OpOperationalLines = OpCodeTable[op_bin].Operational;
147:     OpControlLines = OpCodeTable[op_bin].Control;
148: }
149:
150: // Instruction Decode
151:
152: void T1410CPU::InstructionDecode()
153: {
154:     int op_bin;
155:     BCD b;
156:
157:     // The initial state of the ring gets special handling for NOP
158:
159:     if(IRing -> State() == I_RING_OP) {
160:
161:         BranchToILatch = BranchLatch = false;
162:
163:         // Take an I Cycle
164:
165:         *STAR = *I_AR;
166:         Readout();                                // Next inst. character now in B Register
167:         Cycle();
168:
169:         // In the real machine, what happens is that if a character past the
170:         // op code is read out contained a WM, I_AR is not set, but is then
171:         // later set from STAR. We handle that this way here.
172:
173:         if(!B_Reg -> Get().TestWM()) {
174:             I_AR -> Set(STARMod(+1));
175:         }
176:
177:         // If this is a NOP, it gets special handling right here.
178:         // If there is no wordmark, we do nothing, and come right back
179:         // here with the next cycle (to check for a WM again).
180:         // If there is a WordMark, we decode the op.
181:
182:         if((Op_Reg -> Get() & 0x3f) == OP_NOP) {
183:
184:             if(B_Reg -> Get().TestWM()) {
185:
186:                 // Copy the opcode into the Op register, and decode it.
187:
188:                 *Op_Reg = *B_Reg;
189:
190:                 op_bin = Op_Reg -> Get().ToInt() & 0x3f;
191:                 OpReadOutLines = OpCodeTable[op_bin].ReadOut;
192:                 OpOperationalLines = OpCodeTable[op_bin].Operational;
193:                 OpControlLines = OpCodeTable[op_bin].Control;
194:             }
195:             return;
196:         }
197:
198:         // If there are no control lines set, it is an invalid op code
```

```
199:  
200:        if(OpReadOutLines == OP_INVALID || OpOperationalLines == OP_INVALID ||  
201:            OpControlLines == OP_INVALID) {  
202:            InstructionCheck -> SetStop("Instruction Check: Invalid OP code");  
203:            return;  
204:        }  
205:  
206:        // else Not a NOP. Advance I Ring  
207:  
208:        IRing -> Next();  
209:  
210:        // For % and lozenge ops (that use channel ID), set I Ring to 3  
211:  
212:        if(OpReadOutLines & OP_PERCENTTYPE) {  
213:            IRing -> Set(I_RING_3);  
214:        }  
215:  
216:        // Proceed to "D" on the next cycle  
217:  
218:        return;  
219:    }  
220:  
221:    // At IRing 1 and IRing 6, the index latches are reset.  
222:  
223:    if(IRing -> State() == 1 || IRing -> State() == 6) {  
224:        IndexLatches = 0;  
225:    }  
226:  
227:    // Entry point "D"  
228:  
229:    // If we have a wordmark, handle Instruction length checking...  
230:  
231:    if(B_Reg -> Get().TestWM()) {  
232:  
233:        // Now, since the B Reg has a WM, we need to restore I_AR from  
234:        // STAR, just like in the real machine, so that IAR points to the  
235:        // following opcode. (Otherwise, it would point one past the opcode)  
236:  
237:        *I_AR = *STAR;  
238:  
239:        switch(IRing -> State()) {  
240:  
241:            case I_RING_1:  
242:  
243:                // Handle opcode with no address or which don't need  
244:                // CAR or DAR to chain.  
245:  
246:                if(OpReadOutLines & OP_NOCORDCY) {  
247:                    LastInstructionReadout = true;  
248:                    return;  
249:                }  
250:  
251:                // Handle chaining of arithmetic type op codes  
252:  
253:                if(OpOperationalLines & OP_ARITHTYPE) {  
254:                    CycleRing -> Set(CYCLE_D);  
255:                    *STAR = *B_AR;  
256:                    *D_AR = *STAR; // Mod by 0  
257:  
258:                    // If Multiply or Divide, also set CAR  
259:  
260:                    if(OpOperationalLines & OP_MPYORDIV) {  
261:                        CycleRing -> Set(CYCLE_D);  
262:                        *STAR = *A_AR;  
263:                        *C_AR = *STAR; // Mod by 0  
264:                    }
```

```
265:                     LastInstructionReadout = true;
266:                     return;
267:                 }
268:             }
269:         // Handle chaining of Table Lookup
270:
271:         if((Op_Reg -> Get() & 0x3f) == OP_TABLESEARCH) {
272:             CycleRing -> Set(CYCLE_C);
273:             *STAR = *A_AR;
274:             *C_AR = *STAR;           // Mod by 0
275:             LastInstructionReadout = true;
276:             return;
277:         }
278:     }
279:
280:     // Otherwise, an invalid 1 character opcode
281:
282:     InstructionCheck -> SetStop("Instruction Check: Invalid length at
283:                                   I1");
284:     return;
285:
286:     case I_RING_2:
287:
288:         // Handle simple 2 character op codes
289:
290:         if(OpReadOutLines & OP_2CHARONLY) {
291:             LastInstructionReadout = true;
292:             return;
293:         }
294:
295:         // Otherwise, we have something invalid
296:
297:         InstructionCheck -> SetStop("Instruction Check: Invalid length at
298:                                   I2");
299:         return;
300:
301:     case I_RING_6:
302:
303:         // First, handle ops that end or chain normally with 1 address
304:
305:         if(OpReadOutLines & OP_NODYIRING6) {
306:             LastInstructionReadout = true;
307:             return;
308:         }
309:
310:         // Handle Multiply/Divide chaining
311:
312:         if(OpOperationalLines & OP_MPYORDIV) {
313:             CycleRing -> Set(CYCLE_D);
314:             *STAR = *B_AR;
315:             *C_AR = *STAR;           // Mod by 0
316:             LastInstructionReadout = true;
317:             return;
318:         }
319:
320:         // Otherwise, something is wrong.
321:
322:         InstructionCheck -> SetStop("Instruction Check: Invalid length at
323:                                   I6");
324:         return;
325:
326:     case I_RING_7:
327:
328:         // Handle opcodes that are 1 character plus Op Modifier
329:
330:         if(OpReadOutLines & OP_1ADDRPLUSMOD) {
```

```
328:             LastInstructionReadout = true;
329:             return;
330:         }
331:         // Otherwise, something is wrong
332:         InstructionCheck -> SetStop("Instruction Check: Invalid Length at
333:                                         I7");
334:         return;
335:
336:
337:     case I_RING_11:
338:
339:         // Handle op codes that have 2 addresses with no Op Modifier
340:
341:         if(OpReadOutLines & OP_2ADDRNOMOD) {
342:             LastInstructionReadout = true;
343:             return;
344:         }
345:
346:         // Otherwise, something is wrong.
347:
348:         InstructionCheck -> SetStop("Instruction Check: Invalid Length at
349:                                         I11");
350:         return;
351:
352:     case I_RING_12:
353:
354:         // Handle op codes that have 2 addresses plus an Op Modifier
355:
356:         if(OpReadOutLines & OP_2ADDRPLUSMOD) {
357:             LastInstructionReadout = true;
358:             return;
359:         }
360:
361:         InstructionCheck -> SetStop("Instruction Check: Invalid Length at
362:                                         I12");
363:         return;
364:
365:     default:
366:
367:         InstructionCheck -> SetStop("Instruction Check: Invalid Length");
368:         return;
369:     }
370:
371:     // Make sure we don't fall thru here by accident.
372:
373:     assert(!LastInstructionReadout && !(B_Reg -> Get().TestWM()));
374:
375:     // End of handling of B Channel WordMark
376:
377:     // If this opcode should have addresses, check to make sure that
378:     // the instruction isn't too long...
379:
380:     if(OpReadOutLines & OP_ADDRTYPE) {
381:         if(OpReadOutLines & OP_2ADDRESS) {
382:             if(OpReadOutLines & OP_2ADDRNOMOD) {
383:                 if(IRing -> State() == I_RING_11) {
384:                     InstructionCheck ->
385:                         SetStop("Instruction Check: 2 Addr too long at I11");
386:                     return;
387:                 }
388:                 // OK, proceed to "E"
389:             }
390:         else {
```

```
391:             assert(OpReadOutLines & OP_2ADDRPLUSMOD);
392:             if(IRing -> State() == I_RING_12) {
393:                 InstructionCheck ->
394:                     SetStop("Instruction Check: 2 Addr + Mod too long at
395:                            I12");
396:                 return;
397:             } // OK, Proceed to "E"
398:         }
399:     }
400:     else if(OpReadOutLines & OP_1ADDRPLUSMOD) {
401:         if(IRing -> State() == 7) {
402:             InstructionCheck ->
403:                 SetStop("Instruction Check: 1 Addr + Mod too long at I7");
404:             return;
405:         } // OK, Proceed to "E"
406:     }
407:     else if(IRing -> State() == 6) {
408:         InstructionCheck ->
409:             SetStop("Instruction Check: 1 Addr too long at I6");
410:         return;
411:     }
412:     // OK, Proceed to "E"
413: }
414:
415:
416: else if(OpReadOutLines & OP_2CHARONLY) {
417:
418:     if(IRing -> State() == 1) {
419:         *Op_Mod_Reg = *B_Reg;
420:         InstructionDecodeIARAdvance();
421:         return;
422:     }
423:
424:     else {
425:         InstructionCheck ->
426:             SetStop("Instruction Check: 2 Char Only too long at I2");
427:         return;
428:     }
429: }
430:
431: // ALL ops are either address type or 2 character!
432:
433: else {
434:     InstructionCheck -> SetStop("Instruction Check: Not Addr or 2 Char ???");
435:     return;
436: }
437:
438: // Entry point "E" - no wordmark
439:
440: // First, handle the I/O stuff (Percent type ops)
441:
442: if(OpReadOutLines & OP_PERCENTTYPE) {
443:
444:     // The sections of code doing address validity checking show up
445:     // under entry point "C" on the flowchart, but since this
446:     // section doesn't end up there, we need to do it here.
447:
448:
449:     switch(IRing -> State()) {
450:
451:     case I_RING_3:
452:
453:         if(IOMoveModeLatch || IOLoadModeLatch) {
454:             IOInterlockCheck ->
455:                 SetStop("I/O Interlock Check: I/O in progress at I3");
```

```
456:             return;
457:         }
458:
459:         // IO channel/overlap indicator must have 84 bit configuration
460:
461:         if((B_Reg -> Get() & BIT_NUM) != 0x0c) {
462:             AddressChannelCheck ->
463:                 SetStop("Address Channel Check: I/O Ch/Ovlp must have 84
464: config");
465:             return;
466:
467:             IOChannelSelect = ((B_Reg -> Get() & BITB) != 0);
468:             InstructionDecodeIARAdvance();
469:             return;
470:
471:             case I_RING_4:
472:
473:                 *(Channel[IOChannelSelect] -> ChUnitType) = *B_Reg;
474:                 InstructionDecodeIARAdvance();
475:                 return;
476:
477:             case I_RING_5:
478:
479:                 // Unit number is not allowed to have zones
480:
481:                 if(B_Reg -> Get().ToInt() & BIT_ZONE) {
482:                     AddressChannelCheck ->
483:                         SetStop("Address Channel Check: Zones over unit number");
484:                 }
485:
486:                 *(Channel[IOChannelSelect] -> ChUnitNumber) = *B_Reg;
487:                 InstructionDecodeIARAdvance();
488:                 return;
489:
490:             default:
491:
492:                 assert(IRing -> State() > 5);
493:
494:                 break; // Continue on, knowing that for an I/O op we are
495:                         // in I6 and above, so we will go to step "1" soon.
496:
497:             } // End IRing switch for Percent Type ops
498:
499:         } // End Percent Type ops
500:
501:         // Check to see if we are handling the 2nd address for 2 address
502:         // ops (or the only address for I/O ops) (Step 1 on page 45)
503:
504:         assert(IRing -> State() > 0);
505:
506:         if(IRing -> State() > 5) {
507:
508:             if(OpReadOutLines & OP_1ADDRPLUSMOD) {
509:                 *Op_Mod_Reg = *B_Reg;
510:                 if(OpOperationalLines & OP_BRANCHTYPE) {
511:                     // Branch Handling will go here!
512:                     InstructionDecodeIARAdvance();
513:                     return;
514:                 }
515:                 else {
516:                     InstructionDecodeIARAdvance();
517:                     return;
518:                 }
519:             }
520:
```

```
521:         assert(OpReadOutLines & OP_2ADDRESS);
522:
523:         // For 2 address ops, and for I/O operations,
524:         // snag op modifier at I11.
525:         // (Invalid lengths are checked before we get here).
526:
527:         if(IRing -> State() > I_RING_10) {
528:             *Op_Mod_Reg = *B_Reg;
529:             if(OpOperationalLines & OP_BRANCHTYPE) {
530:                 // Branch Handling will go here!
531:                 InstructionDecodeIARAdvance();
532:                 return;
533:             }
534:             else {
535:                 InstructionDecodeIARAdvance();
536:                 return;
537:             }
538:         }
539:
540:         // If we are at I6, we are starting address: reset BAR, DAR
541:
542:         if(IRing -> State() == I_RING_6) {
543:             B_AR -> Reset();
544:             D_AR -> Reset();
545:         }
546:
547:         // Set the appropriate address character into B & DAR
548:
549:         b = B_Reg -> Get();
550:         b = b & BIT_NUM;
551:         b.SetOddParity();
552:
553:         B_AR -> Set(TWOOF5(b),IRing -> State() - 5);
554:         D_AR -> Set(TWOOF5(b),IRing -> State() - 5);
555:
556:         if(IRing -> State() == I_RING_10) {
557:             InstructionIndexStart();           // Start up indexing
558:             return;
559:         }
560:
561:         // Fall thru to "C"
562:
563:     } // Ending IRing > 5
564:
565:     // Handle IRings 1 thru 5.
566:
567: else {
568:
569:     if(OpReadOutLines & OP_ADDRDBL) {
570:
571:         if(IRing -> State() == 1) {
572:             A_AR -> Reset();
573:             B_AR -> Reset();
574:             C_AR -> Reset();
575:             D_AR -> Reset();
576:         }
577:
578:         b = B_Reg -> Get();
579:         b = b & BIT_NUM;
580:         b.SetOddParity();
581:
582:         A_AR -> Set(TWOOF5(b),IRing -> State());
583:         B_AR -> Set(TWOOF5(b),IRing -> State());
584:         C_AR -> Set(TWOOF5(b),IRing -> State());
585:         D_AR -> Set(TWOOF5(b),IRing -> State());
586:
```

```
587:         if(IRing -> State() == I_RING_5) {           // Start up indexing
588:             InstructionIndexStart();                   // Start up indexing
589:             return;
590:         }
591:         // Fall thru to step "C"
592:     }
593:
594:
595:     // This coding varies slightly from the flowchart on page 45.
596:     // The logic is the same, but we test for SAR (opcode G) first
597:     // to avoid redundant code.
598:
599:     // It works alot easier this way because SAR has two special features:
600:     // It doesn't reset AAR (so you can store AAR), and it cannot be
601:     // indexed.
602:
603:     else if((Op_Reg -> Get() & 0x3f) == OP_SAR_G) {      // SAR
604:
605:         if(IRing -> State() == I_RING_1) {
606:             C_AR -> Reset();
607:         }
608:
609:         b = B_Reg -> Get();
610:         b = b & BIT_NUM;
611:         b.SetOddParity();
612:
613:         C_AR -> Set(TWOOF5(b),IRing -> State());
614:     }
615:
616:     else {        // Not SAR
617:
618:         if(IRing -> State() == I_RING_1) {
619:             A_AR -> Reset();
620:             C_AR -> Reset();
621:         }
622:
623:         b = B_Reg -> Get();
624:         b = b & BIT_NUM;
625:         b.SetOddParity();
626:
627:         A_AR -> Set(TWOOF5(b),IRing -> State());
628:         C_AR -> Set(TWOOF5(b),IRing -> State());
629:
630:         if(IRing -> State() == I_RING_5 &&
631:             IndexLatches != 0) {
632:             InstructionIndexStart();           // Start up indexing
633:             return;
634:         }
635:
636:     }
637:
638: }        // End of I Ring 1 - 5
639:
640:
641: // Entry point "C" - address validity checking.
642: // Odd that this happens in the flow chart *after* setting the character
643: // into the address register. Oh well - the 2-of-5 translater is robust!
644: // It sets invalid entries to 0.
645:
646: // First, handle special characters. They are only allowed on I/O
647: // (Percent Type) opcodes
648:
649: if(B_Reg -> Get().GetType() == BCD_SC) {
650:     if((OpReadOutLines & OP_PERCENTTYPE) == 0) {
651:         AddressChannelCheck ->
652:             SetStop("Address Channel Check: Special Chars, not % type op");
```

```
653:             return;
654:         }
655:     }
656:
657: // Next, check for zones, which are only valid at certain times
658:
659: if(B_Reg -> Get().ToInt() & BIT_ZONE) {
660:     if(!IRingZoneTable[IRing -> State()]) {
661:         AddressChannelCheck ->
662:             SetStop("Address Channel Check: Zones at invalid IRing time");
663:         return;
664:     }
665:
666: // Set the index latches
667:
668: IndexLatches |= (B_Reg -> Get().ToInt() & BIT_ZONE) >>
669:     ((IRing -> State() == I_RING_3 || IRing -> State() == I_RING_8)
670:      ? 2 : 4);
671:
672: }
673:
674: // Step "B": Read out next character, advance I-Ring
675: // Will pick up again at step "D"
676:
677: InstructionDecodeIARAdvance();
678:
679: }
680:
681: // Routine to implement Step "B" in the chart. It takes an I cycle
682: // (reads out character pointed to by I_AR), advances IRing, and, if
683: // the newly read character doesn't have a word mark, advances I_AR.
684:
685: // In the real 1410, what happens with the I_AR is that the WM inhibits
686: // setting the I_AR from STAR+1, and then later causes I_AR to be copied
687: // from the STAR (which has the old address).
688:
689: void T1410CPU::InstructionDecodeIARAdvance()
690: {
691:     *STAR = *I_AR;
692:     IRing -> Next();
693:     Readout();
694:     Cycle();
695:     if(!B_Reg -> Get().TestWM()) {
696:         I_AR -> Set(STARMod(+1));
697:     }
698:     return;
699: }
700:
701: // Routine to start up indexing.
702: // When an index is present at IRing 5 or IRing 11 times, we do this instead
703: // of advancing the IRing. This then causes X Cycles (see below)
704:
705: void T1410CPU::InstructionIndexStart()
706: {
707:     CycleRing -> Set(CYCLE_X); // Doing X cycles
708:     ARing -> Reset(); // Reset A Ring to initial state
709:
710:     assert(IndexLatches > 0 && IndexLatches < 16);
711:
712: // Use the "address generator" to address the proper index register
713:
714: STAR -> Set(IndexRegisterLookup[IndexLatches]);
715:
716: // Advance to A2, and read out first index register character
717: // Address modification by -1 for this sycle
718:
```

```
719:     ARing -> Next();
720:     Readout();
721:     Cycle();                                // Does nothing
722:     STAR -> Set(STARMod(-1));
723: }
724:
725: // Indexing routine
726:
727: void T1410CPU::InstructionIndex()
728: {
729:     BCD sum;
730:
731:     assert(IRing -> State() == I_RING_5 || IRing -> State() == I_RING_10);
732:
733:     if(IRing -> State() == I_RING_5) {
734:         if(ARing -> State() == A_RING_2) {
735:             if(OpReadOutLines & OP_ADDRDBL) {
736:                 B_AR -> Reset();
737:                 D_AR -> Reset();
738:             }
739:             else {
740:                 A_AR -> Reset();
741:             }
742:         }
743:
744:         A_Reg -> Set(C_AR -> GateBCD(6 - (ARing -> State()))); // Gate C Address
745:     reg.
746:     }
747:     else {
748:         if(ARing -> State() == A_RING_2) {
749:             B_AR -> Reset();
750:         }
751:
752:         A_Reg -> Set(D_AR -> GateBCD(6 - (ARing -> State()))); // Gate D Address
753:     reg.
754:     }
755:
756:     // Determine sign of indexing...
757:
758:     if(ARing -> State() == A_RING_2) {
759:         BComplement -> Set(B_Reg -> Get().IsMinus());
760:         CarryIn -> Reset();
761:
762:         // Run it thru the adder, with the A channel coming from A Register
763:         Adder(AChannel -> Select(AChannel -> A_Channel_A), false,
764:               B_Reg -> Get(), BComplement -> State());
765:
766:         DEBUG("Indexing. Added %x", AChannel -> Select().ToInt());
767:         DEBUG("      And    %x", B_Reg -> Get().ToInt());
768:         DEBUG("      SUM    %x", AdderResult.ToInt());
769:
770:         // Gate adder numerics to assembly channel
771:
772:         sum = AssemblyChannel -> Select(
773:             AssemblyChannel -> AsmChannelWMNone,
774:             AssemblyChannel -> AsmChannelZonesNone,
775:             false,
776:             AssemblyChannel -> AsmChannelSignNone,
777:             AssemblyChannel -> AsmChannelNumAdder );
778:
779:         if(IRing -> State() == 5) {
780:             if(OpReadOutLines & OP_ADDRDBL) {
781:                 B_AR -> Set(TWOOF5(sum), 6 - (ARing -> State()));
782:             }
783:         }
784:     }
785:
```

```
783:         A_AR -> Set(TWOOF5(sum), 6 - (ARing -> State()));
784:     }
785: else {
786:     B_AR -> Set(TWOOF5(sum), 6 - (ARing -> State()));
787:     D_AR -> Set(TWOOF5(sum), 6 - (ARing -> State()));
788: }
789:
790: // If indexing operation is done, set registers right, and restart
791: // the instruction readout process
792:
793: if(ARing -> State() == A_RING_6) {
794:     if(IRing -> State() == I_RING_5) {
795:         *C_AR = *A_AR;
796:     }
797:     else {
798:         *D_AR = *B_AR;
799:     }
800:     CycleRing -> Set(CYCLE_I); // Doing I cycles again
801:     InstructionDecodeIARAdvance();
802:     return;
803: }
804:
805: // Otherwise, advance to the next X Cycle
806:
807: ARing -> Next();
808: Readout();
809: Cycle();
810: STAR -> Set(STARMod(-1));
811: }
812:
```

```
1: //-----
2: #ifndef UI1410DEBUGH
3: #define UI1410DEBUGH
4: //-----
5: #include <vcl\Classes.hpp>
6: #include <vcl\Controls.hpp>
7: #include <vcl\StdCtrls.hpp>
8: #include <vcl\Forms.hpp>
9: //-----
10:
11: #define DEBUG(string,val) \
12:     sprintf(F1410Debug->line,string,val); \
13:     F1410Debug -> DebugOut(F1410Debug -> line);
14:
15:
16: class TF1410Debug : public TForm
17: {
18:     __published:    // IDE-managed Components
19:         TMemo *Debug;
20:     private:        // User declarations
21:     public:         // User declarations
22:         __fastcall TF1410Debug(TComponent* Owner);
23:         char line[256];
24:         void __fastcall DebugOut(char *s);    Review ✓
25: };
26: //-----
27: extern TF1410Debug *F1410Debug;
28: //-----
29: #endif
```

```
1: //-----
2: #include <vcl.h>
3: #pragma hdrstop
4:
5: #include "UI1410DATA.h"
6:
7: //-----
8: #pragma package(smart_init)
9:
10: #include <assert.h>
11:
12: #include "UI1410CPUUT.H"
13: #include "UI1410INST.H"
14: #include "UI1410DEBUG.H"
15:
16: int edit_char_flag [64] = {
17:     SZ_SUPPRESS_IF_ZERO |           /* 0          - space */
18:     EDIT_UNITS_SPECIAL |           /* 1          - 1 */
19:     EDIT_BODY_SPECIAL |           /* 2          - 2 */
20:     EDIT_SUPPRESS_IF_NS,           /* 3          - 3 */
21:     SZ_SIGNIFICANT_DIGIT,         /* 4          - 4 */
22:     SZ_SIGNIFICANT_DIGIT,         /* 5          - 5 */
23:     SZ_SIGNIFICANT_DIGIT,         /* 6          - 6 */
24:     SZ_SIGNIFICANT_DIGIT,         /* 7          - 7 */
25:     SZ_SIGNIFICANT_DIGIT,         /* 8          - 8 */
26:     SZ_SIGNIFICANT_DIGIT,         /* 9          - 9 */
27:     SZ_SUPPRESS_IF_ZERO |           /* 10         - 0 */
28:     EDIT_UNITS_SPECIAL |           /* 11         - number sign (#) or equal */
29:     EDIT_BODY_SPECIAL |           /* 12         - at sign @ or quote */
30:     EDIT_SUPPRESS_IF_NS,           /* 13         - colon */
31:     0,                           /* 14         - greater than */
32:     0,                           /* 15         - radical */
33:     0,                           /* 16         - substitute blank */
34:     0,                           /* 17         - slash */
35:     0,                           /* 18         - S */
36:     0,                           /* 19         - T */
37:     0,                           /* 20         - U */
38:     0,                           /* 21         - V */
39:     0,                           /* 22         - W */
40:     0,                           /* 23         - X */
41:     0,                           /* 24         - Y */
42:     0,                           /* 25         - Z */
43:     0,                           /* 26         - record mark */
44:
45:     SZ_SUPPRESS_IF_ZERO |           /* 27         - comma */
46:     EDIT_EXT_SPECIAL |           /* 28         - percent % or paren */
47:     EDIT_SUPPRESS_IF_NS,           /* 29         - word separator */
48:     0,                           /* 30         - left oblique */
49:     0,                           /* 31         - segment mark */
50:
51:     SZ_INCLUDE_IF_ZERO |           /* 32         - hyphen */
52:     EDIT_SUPPRESS_IF_PLUS |           /* 33         - J */
53:     EDIT_UNITS_SPECIAL |           /* 34         - K */
54:     EDIT_EXT_SPECIAL |           /* 35         - L */
55:     EDIT_SUPPRESS_IF_NS,           /* 36         - M */
56:     0,                           /* 37         - N */
57:
58:
59:
60:
61:
62:
63:
64:
65:
66:
```

```

67:          0,                  /* 38 B 42 - O */
68:          0,                  /* 39 B 421 - P */
69:          0,                  /* 40 B 8 - Q */
70:          EDIT_SUPPRESS_IF_PLUS |
71:          EDIT_UNITS_SPECIAL |
72:          EDIT_EXT_SPECIAL,    /* 41 B 8 1 - R */
73:          0,                  /* 42 B 8 2 - exclamation (-0) */
74:          EDIT_BODY_SPECIAL,   /* 43 B 8 21 - dollar sign */
75:          EDIT_BODY_SPECIAL,   /* 44 B 84 - asterisk */
76:          0,                  /* 45 B 84 1 - right bracket */
77:          0,                  /* 46 B 842 - semicolon */
78:          0,                  /* 47 B 8421 - delta */
79:          EDIT_UNITS_SPECIAL |
80:          EDIT_EXT_SPECIAL |
81:          EDIT_BODY_SPECIAL,   /* 48 BA - ampersand or plus */
82:          0,                  /* 49 BA 1 - A */
83:          0,                  /* 50 BA 2 - B */
84:          EDIT_SUPPRESS_IF_PLUS |
85:          EDIT_UNITS_SPECIAL |
86:          EDIT_EXT_SPECIAL,    /* 51 BA 21 - C */
87:          0,                  /* 52 BA 4 - D */
88:          0,                  /* 53 BA 4 1 - E */
89:          0,                  /* 54 BA 42 - F */
90:          0,                  /* 55 BA 421 - G */
91:          0,                  /* 56 BA8 - H */
92:          0,                  /* 57 BA8 1 - I */
93:          0,                  /* 58 BA8 2 - question mark */
94:          SZ_INCLUDE_IF_ZERO |
95:          EDIT_SUPPRESS_IF_NS,  /* 59 BA8 21 - period */
96:          0,                  /* 60 BA84 - lozenge or paren */
97:          0,                  /* 61 BA84 1 - left bracket */
98:          0,                  /* 62 BA842 - less than */
99:          0,                  /* 63 BA8421 - group mark */

100: };
101:
102: OK          AC OK
103: void T1410CPU::InstructionMove()
104: {
105:
106:     static struct {
107:         char cycle;
108:         char subcycle;
109:         char scan;
110:         char subscan;
111:     } next;
112:
113:     static int op_mod_bin;
114:     BCD b_temp;
115:
116:     enum TAssemblyChannel::AsmChannelNumericSelect AsmChannelNumericSelect;
117:     enum TAssemblyChannel::AsmChannelZonesSelect AsmChannelZonesSelect;
118:     enum TAssemblyChannel::AsmChannelWMSelect AsmChannelWMSelect;
119:
120:     if>LastInstructionReadout) {
121:         op_mod_bin = Op_Mod_Reg -> Get().ToInt();
122:         next.scan = (op_mod_bin & BIT8) ? SCAN_2 : SCAN_1;
123:         ScanRing -> Set(next.scan);
124:         next.subscan = SUB_SCAN_U;
125:         SubScanRing -> Set(next.subscan);
126:         next.cycle = CYCLE_A;
127:         next.subcycle = 0;
128:     }
129:
130: // Do common items for all cycles
131:
132:     CycleRing -> Set(next.cycle);

```

```

133:     ScanRing -> Set(next.scan);
134:     SubScanRing -> Set(next.subscan);
135:
136:     // The "if" statements use the next variable for easy coding
137:
138:     if(next.cycle == CYCLE_A && next.subcycle == 0) {
139:         *STAR = *A_AR;
140:         Readout();                                // Readout A field char
141:         Cycle();
142:         next.cycle = CYCLE_B;                    // B Cycle next
143:         next.subcycle = 0;
144:         return;
145:     }
146:
147:     if(next.cycle == CYCLE_B && next.subcycle == 0) {
148:         *STAR = *B_AR;
149:         Readout();                                // Readout B field char
150:         OK b_temp = B_Reg -> Get();           // Save B before move
151:         AsmChannelNumericSelect = (op_mod_bin & BIT1) ?
152:             AssemblyChannel -> AsmChannelNumA :
153:             AssemblyChannel -> AsmChannelNumB;
154:         AsmChannelZonesSelect = (op_mod_bin & BIT2) ?
155:             AssemblyChannel -> AsmChannelZonesA :
156:             AssemblyChannel -> AsmChannelZonesB;
157:         AsmChannelWMSelect = (op_mod_bin & BIT4) ?
158:             AssemblyChannel -> AsmChannelWMA :
159:             AssemblyChannel -> AsmChannelWMB;
160:         Store(AssemblyChannel -> Select (           // Store the result
161:             AsmChannelWMSelect,
162:             AsmChannelZonesSelect,
163:             false,
164:             AssemblyChannel -> AsmChannelSignNone,
165:             AsmChannelNumericSelect ) );
166:
167:         Cycle();
168:         next.cycle = CYCLE_A;                      // For now, assume we
169:         next.subcycle = 0;                          // will continue on
170:         next.subscan = SUB_SCAN_B;                  // next will be body
171:         // Regen Scan Ctrl (1 or 2)
172:
173:         // The next bits set IRingControl if we should stop here...
174:
175:         if( !(op_mod_bin & (BIT8 | BITA | BITB)) ) { // None of 8, A, B
176:             IRingControl = true;                     // single char - done
177:         }
178:         else if( !(op_mod_bin & (BIT8)) ) {        // No 8 bit - R to L
179:             if( ((op_mod_bin & BITA) && A_Reg -> Get().TestWM()) ||
180:                 ((op_mod_bin & BITB) && b_temp.TestWM()) ) {
181:                 IRingControl = true; OK                // Stop on approp. WM
182:             }
183:         }
184:         else if( !(op_mod_bin & (BITA | BITB)) ) { // 8, not A or B
185:             IRingControl = (A_Reg -> Get().TestWM() || b_temp.TestWM()); OK
186:         }
187:         else if(op_mod_bin & (BITA | BITB)) {
188:             IRingControl = ((op_mod_bin & BITA) && A_Reg -> Get().TestRM() ) ||
189:                             ((op_mod_bin & BITB) && A_Reg -> Get().TestGMWM() );
190:         }
191:         else {
192:             assert(false);                           // Should not get here
193:         }
194:
195:         return;
196:     } // End B Cycle, subtype 0
197:
198: }

```

```
199:  
200: void T1410CPU::InstructionMoveSuppressZeros()  
201: {  
202:  
203:     static struct {  
204:         char cycle;  
205:         char subcycle;  
206:         char scan;  
207:         char subscan;  
208:     } next;  
209:  
210:     enum TAssemblyChannel::AsmChannelNumericSelect AsmChannelNumericSelect;  
211:     enum TAssemblyChannel::AsmChannelWMSelect AsmChannelWMSelect;  
212:     enum TAssemblyChannel::AsmChannelZonesSelect AsmChannelZonesSelect;  
213:  
214:     int sz_char_flags;  
215:     BCD b_temp;  
216:  
217:     if(LastInstructionReadout) {  
218:         next.scan = SCAN_1;  
219:         ScanRing -> Set(next.scan);  
220:         next.subscan = SUB_SCAN_U;  
221:         SubScanRing -> Set(next.subscan);  
222:         next.cycle = CYCLE_A;  
223:         next.subcycle = 0;  
224:         ZeroSuppressLatch = false;  
225:     }  
226:  
227:     // Do common items for all cycles  
228:  
229:     CycleRing -> Set(next.cycle);  
230:     ScanRing -> Set(next.scan);  
231:     SubScanRing -> Set(next.subscan);  
232:  
233:     // The "if" statements use the "next" variable for easy coding  
234:  
235:     if(next.cycle == CYCLE_A && next.subcycle == 0) {  
236:         *STAR = *A_AR;  
237:         Readout();                                // Read out A field char  
238:         Cycle();  
239:         next.cycle = CYCLE_B;  
240:         next.subcycle = 0;  
241:         return;  
242:     } // End A Cycle  
243:  
244:     if(next.cycle == CYCLE_B && next.subcycle == 0) {  
245:         *STAR = *B_AR;  
246:         Readout();                                // Read out B field char  
247:  
248:         if(SubScanRing -> State() == SUB_SCAN_U) {    // Units?  
249:             AsmChannelZonesSelect = AssemblyChannel -> AsmChannelZonesNone;  
250:             AsmChannelWMSelect = AssemblyChannel -> AsmChannelWMSet;  
251:             ZeroSuppressLatch = true;  
252:         }  
253:         else {                                         // No (body)  
254:             assert(SubScanRing -> State() == SUB_SCAN_B);  
255:             AsmChannelZonesSelect = AssemblyChannel -> AsmChannelZonesA;  
256:             AsmChannelWMSelect = AssemblyChannel -> AsmChannelWMNone;  
257:         }  
258:  
259:         AsmChannelNumericSelect = AssemblyChannel -> AsmChannelNumA;  
260:  
261:         // Store / AssemblyChannel -> Select (   
262:         AsmChannelWMSelect,  
263:         AsmChannelZonesSelect,  
264:         false,
```

```

265:             AssemblyChannel -> AsmChannelSignNone,
266:             AsmChannelNumericSelect ) );
267:
268:     Cycle();
269:
270:     if(A_Reg -> Get().TestWM()) { // AChannel WM?
271:         next.cycle = CYCLE_B; // Yes. Start 2nd scan
272:         next.subcycle = 1;
273:         next.scan = SCAN_2;
274:         next.subscan = SUB_SCAN_MQ; // Set MQ latch for Skid
275:         return;
276:     }
277:     else { // No A Channel WM
278:         next.cycle = CYCLE_A; // A cycle next
279:         next.subcycle = 0; // First scan
280:         assert(ScanRing -> State() == SCAN_1); // Regen 1st scan
281:         next.subscan = SUB_SCAN_B; // Set Body Latch
282:         return;
283:     }
284: } // End B Cycle, subtype 0
285:
286: if(next.cycle == CYCLE_B && next.subcycle == 1) {
287:     *STAR = *B_AR;
288:     Readout(); // RO B field char
289:     if(SubScanRing -> State() == SUB_SCAN_MQ) { // Skid Cycle (MQ) ?
290:         next.subscan = SUB_SCAN_E; // Yes. Extension next
291:         ✓ OK ✓ Store( AssemblyChannel -> Select // AssemblyChannel -> AsmChannelWMB, // B Ch WM
292:                     AssemblyChannel -> AsmChannelZonesB, // B Character
293:                     false,
294:                     AssemblyChannel -> AsmChannelSignNone,
295:                     AssemblyChannel -> AsmChannelNumB );
296:         Cycle();
297:         assert(ScanRing -> State() == SCAN_2); // Regen 2nd scan
298:         return; // Next cycle same kind
299:     }
300: }
301:
302: ✓ OK ✓ Store( AssemblyChannel -> Select // Assume we will store B
303: sz_char_flags = edit_char_flag[B_Reg -> Get().ToInt() & 0x3f];
304: if(sz_char_flags & SZ_SIGNIFICANT_DIGIT) {
305:     ZeroSuppressLatch = false;
306: }
307: else if(!(sz_char_flags & (SZ_INCLUDE_IF_ZERO | SZ_SUPPRESS_IF_ZERO))) {
308:     // NOT: , 0 sp . -
309:     ZeroSuppressLatch = true;
310: }
311: else if(sz_char_flags & SZ_INCLUDE_IF_ZERO) {
312:     // . or - will just Store B w/o WM (later)
313: }
314: else {
315:     assert(sz_char_flags & SZ_SUPPRESS_IF_ZERO); // Must be sp , 0
316:     if(ZeroSuppressLatch) {
317:         B_Reg -> Set(BITC); // AssemblyChannel → Set (BITC);
318:     }
319: }
320:
321: // b_temp has original B Channel character!
322: Store( AssemblyChannel -> Select (
323:             AssemblyChannel -> AsmChannelWMNone, // Store, no WM
324:             AssemblyChannel -> AsmChannelZonesB, // B Char
325:             false,
326:             AssemblyChannel -> AsmChannelSignNone,
327:             AssemblyChannel -> AsmChannelNumB );
328:
329: Cycle();
330:
}

```

restore B reg.

text  
Special Char

B\_Re → Get()

```

331:     if(b_temp.TestWM()) {                                // Did org. B WM?
332:         IRingControl = true;                            // If so, done
333:         return;
334:     }
335:     else {
336:         assert(SubScanRing -> State() == SUB_SCAN_E); // Regen Extension
337:         assert(ScanRing -> State() == SCAN_2);        // Regen 2nd scan
338:         // Next cycle same kind.
339:         return;
340:     }
341: } // End B Cycle, subtype 1
342:
343: } // End, Move and Suppress Zeros
344:
345: void T1410CPU::InstructionEdit()
346: {
347:     static struct {
348:         char cycle;
349:         char subcycle;
350:         char scan;
351:         char subscan;
352:     } next;
353:     ↗
354:     bool FirstBFieldZero = false;
355:     int edit_char_flags;
356:     int first_scan_store_type = 0;
357:     BCD b_temp;
358:
359:     enum TAassemblyChannel::AsmChannelZonesSelect AsmChannelZonesSelect;
360:     enum TAassemblyChannel::AsmChannelWMSelect AsmChannelWMSelect;
361:
362:     if>LastInstructionReadout) {
363:         next.scan = SCAN_1;
364:         ScanRing -> Set(next.scan);
365:         next.subscan = SUB_SCAN_U;
366:         SubScanRing -> Set(next.subscan);
367:         next.cycle = CYCLE_A;
368:         next.subcycle = 0;
369:         SignLatch = false;
370:         FloatingDollarLatch = false;
371:         AsteriskFillLatch = false;
372:         DecimalControlLatch = false;
373:     }
374:
375:     // Do common items for all cycles
376:
377:     CycleRing -> Set(next.cycle);
378:     ScanRing -> Set(next.scan);
379:     SubScanRing -> Set(next.subscan);
380:
381:     // The "if" statements use the next variable for easy coding
382:
383:     if(next.cycle == CYCLE_A && next.subcycle == 0) {
384:         *STAR = *A_AR;
385:         Readout();                                         // Readout A char
386:         if(SubScanRing -> State() == SUB_SCAN_U) {
387:             SignLatch = Reg -> Get().IsMinus();          // Determine sign
388:         }
389:         ↗
390:         next.cycle = CYCLE_B;
391:         next.subcycle = 0;
392:         return;
393:     } // End A Cycle, subtype 0
394:
395:     if(next.cycle == CYCLE_B && next.subcycle == 0) {
396:         *STAR = *B_AR;

```

// While still in B Reg

```

397:     Readout();                                     // Readout B field char
398:     b_temp = B_Reg -> Get();
399:
400:     if(b_temp.TestChar(BCD_0) && !ZeroSuppressLatch) {
401:         FirstBFieldZero = ZeroSuppressLatch = true;
402:     }
403:
404:     edit_char_flags = edit_char_flag[b_temp.ToInt() & 0x3f];
405:
406:     if(SubScanRing -> State() == SUB_SCAN_U) {      // Units
407:         if(edit_char_flags & EDIT_UNITS_SPECIAL) {
408:             if((edit_char_flags & EDIT_SUPPRESS_IF_PLUS) && !SignLatch) ||
409:                 b_temp.TestChar(BCD_AMPERSAND)) {
410:                     first_scan_store_type = 2;
411:                 }
412:                 else if(b_temp.TestChar(BCD_SPACE) || b_temp.TestChar(BCD_0)) {
413:                     first_scan_store_type = 4;
414:                 }
415:                 else {
416:                     assert((edit_char_flags & EDIT_SUPPRESS_IF_PLUS) && SignLatch);
417:                     first_scan_store_type = 1;
418:                 }
419:             } // End Units Special
420:             else {
421:                 first_scan_store_type = 1;
422:             }
423:         } // End Units
424:
425:         else if(SubScanRing -> State() == SUB_SCAN_E) { // Extension
426:             if(edit_char_flags & EDIT_EXT_SPECIAL) {
427:                 if((edit_char_flags & EDIT_SUPPRESS_IF_PLUS) && !SignLatch) ||
428:                     b_temp.TestChar(BCD_AMPERSAND) ||
429:                     b_temp.TestChar(BCD_COMMA)) {
430:                         first_scan_store_type = 2;
431:                     }
432:                     else {
433:                         assert((edit_char_flags & EDIT_SUPPRESS_IF_PLUS) && SignLatch);
434:                         first_scan_store_type = 1;
435:                     }
436:                 } // End Ext. Special
437:                 else {
438:                     first_scan_store_type = 1;
439:                 }
440:             } // End Extension
441:
442:             else {
443:                 assert(SubScanRing -> State() == SUB_SCAN_B); // Body
444:                 if(edit_char_flags & EDIT_BODY_SPECIAL) {
445:                     if(b_temp.TestChar(BCD_AMPERSAND)) {
446:                         first_scan_store_type = 2;
447:                     }
448:                     else if(b_temp.TestChar(BCD_0) || b_temp.TestChar(BCD_SPACE)) {
449:                         first_scan_store_type = 3;
450:                     }
451:                     else {
452:                         assert(b_temp.TestChar(BCD_DOLLAR) ||
453:                               b_temp.TestChar(BCD_ASTERISK));
454:                         if(ZeroSuppressLatch && !FloatingDollarLatch &&
455:                             !AsteriskFillLatch) {
456:                             FloatingDollarLatch = b_temp.TestChar(BCD_DOLLAR);
457:                             AsteriskFillLatch = b_temp.TestChar(BCD_ASTERISK);
458:                         }
459:                         first_scan_store_type = 3;
460:                     }
461:                 } // End Body Special

```

```

463:         else {
464:             first_scan_store_type = 1;
465:         }
466:     } // End Body
467:
468:     if(first_scan_store_type == 1 || first_scan_store_type == 2) {
469:         if(first_scan_store_type == 2) { // Point 2: Store a blank
470:             B_Reg -> Set(BCD_SPACE); AssemblyChannel → Set(BCD-Space);
471:             AsmChannelWMSelect = TAssemblyChannel::AsmChannelWMNone;
472:         }
473:         else {
474:             assert(first_scan_store_type == 1); // Point 1: Store B + W
475:             AsmChannelWMSelect = TAssemblyChannel::AsmChannelWMSet;
476:         }
477:         Store( AssemblyChannel -> Select (
478:             AsmChannelWMSelect,
479:             TAssemblyChannel::AsmChannelZonesB,
480:             false,
481:             TAssemblyChannel::AsmChannelSignNone,
482:             TAssemblyChannel::AsmChannelNumB ) );
483:         Cycle(); OK
484:         if(b_temp.TestWM()) { // B Channel WM? Yes...
485:             if(ZeroSuppressLatch) { // Zeros found? Yes...
486:                 next.scan = SCAN_2; // Enter 2nd Scan
487:                 next.subscan = SUB_SCAN_MQ; // Enter MQ (skid) cycle
488:                 next.cycle = CYCLE_B; // B Cycle Next
489:                 next.subcycle = 1; // Second kind.
490:             }
491:             else {
492:                 IRingControl = True; // No Zeros found. All done
493:             }
494:         }
495:         else { // Not B Channel WM
496:             assert(ScanRing -> State() == SCAN_1); // Regen 1st scan
497:             next.cycle = CYCLE_B; // More B Cycles
498:             next.subcycle = 0; // Still 1st scan
499:         }
500:         return;
501:     } // End flowchart symbols 1 and 2
502:
503:     else if(first_scan_store_type == 3 || first_scan_store_type == 4) {
504:         AsmChannelWMSelect = FirstBFieldZero ?
505:             TAssemblyChannel::AsmChannelWMSet :
506:             TAssemblyChannel::AsmChannelWMNone;
507:         AsmChannelZonesSelect = (first_scan_store_type == 3) ?
508:             TAssemblyChannel::AsmChannelZonesA : // Point 3: Store A Chr
509:             TAssemblyChannel::AsmChannelZonesNone; // Point 4: Store A Num
510:         Store ( AssemblyChannel -> Select (
511:             AsmChannelWMSelect,
512:             AsmChannelZonesSelect,
513:             false,
514:             TAssemblyChannel::AsmChannelSignNone,
515:             TAssemblyChannel::AsmChannelNumA ) );
516:         Cycle(); OK
517:         if(b_temp.TestWM()) { // B field WM? Yes...
518:             if(ZeroSuppressLatch) { // Zero Found? Yes...
519:                 next.cycle = CYCLE_B; // Start 2nd scan
520:                 next.subcycle = 1;
521:                 next.scan = SCAN_2;
522:                 next.subscan = SUB_SCAN_MQ; // Set for skid cycle
523:             }
524:             else { // No zero found
525:                 IRingControl = true; // All done
526:             }
527:         }
528:     } // No B field WM

```

```

529:         assert(ScanRing -> State() == SCAN_1); // Regen 1st scan
530:         if(A_Reg -> Get().TestWM()) { // A field WM? Yes...
531:             next.subscan = SUB_SCAN_E; // Set Extension
532:             next.cycle = CYCLE_B; // Continue 1st scan
533:             next.subcycle = 0;
534:         }
535:         else { // No A field WM
536:             next.subscan = SUB_SCAN_B; // Set Body
537:             next.cycle = CYCLE_A; // A Cycle next
538:             next.subcycle = 0; // First scan
539:         }
540:     }
541:     return;
542: } // End flowchart symbols 3 and 4
543:
544: else {
545:     assert(false); // Logic error (sb 1-4)
546:
547:
548: } // End B Cycle, subtype 0
549:
550: if(next.cycle == CYCLE_B && next.subcycle == 1) {
551:     *STAR = *B_AR;
552:     Readout(); // Read out B Field chr
553:     b_temp = B_Reg -> Get();
554:     edit_char_flags = edit_char_flag[b_temp.ToInt() & 0x3f];
555:     if(SubScanRing -> State() == SUB_SCAN_MQ) { // In MQ (skid) cycle?
556:         next.cycle = CYCLE_B; // B Cycle Next
557:         next.subcycle = 1; // Same station
558:         next.subscan = SUB_SCAN_E; // Set Extension
559:         ✓ Store (AssemblyChannel -> Select , // Store B Field char
560:          ✓ TAassemblyChannel::AsmChannelWMB,
561:          TAassemblyChannel::AsmChannelZonesB,
562:          false,
563:          TAassemblyChannel::AsmChannelSignNone,
564:          TAassemblyChannel::AsmChannelNumB );
565:         Cycle();
566:         assert(ScanRing -> State() == SCAN_2); // Regen 2nd scan
567:         return;
568:     } // End MQ (Skid) cycle
569:     if(edit_char_flags & SZ_SIGNIFICANT_DIGIT) { Ø - 9
570:         ZeroSuppressLatch = false;
571:     }
572:     else_if(!edit_char_flags & EDIT_SUPPRESS_IF_NS) {
573:         if(!DecimalControlLatch) {
574:             ZeroSuppressLatch = true;
575:         }
576:     }
577:     else_if(b_temp.TestChar(BCD_MINUS)) {
578:     }
579:     else_if(b_temp.TestChar(BCD_PERIOD)) {
580:         if(ZeroSuppressLatch) {
581:             DecimalControlLatch = true;
582:         }
583:     }
584:     else {
585:         assert(b_temp.TestChar(BCD_SPACE) || // Assembly Channel → Set
586:             b_temp.TestChar(BCD_COMMA) || b_temp.TestChar(BCD_0));
587:         if(ZeroSuppressLatch && !DecimalControlLatch) {
588:             B_Reg -> Set(AsteriskFillLatch ? BCD_ASTERISK : BCD_SPACE);
589:         }
590:     }
591: }
592: Store(AssemblyChannel -> Select (
593:     TAassemblyChannel::AsmChannelWMNone,
594:     TAassemblyChannel::AsmChannelZonesB,

```

```

595:         false,
596:         TAssemblyChannel::AsmChannelSignNone,
597:         TAssemblyChannel::AsmChannelNumB ) );
598:     Cycle();
599:
600:     if(!b_temp.TestWM()) {
601:         assert(SubScanRing -> State() == SUB_SCAN_E); // Regen Ext. Latch
602:         assert(ScanRing -> State() == SCAN_2); // Regen 2nd Scan
603:         next.cycle = CYCLE_B; // B Cycle next
604:         next.subcycle = 1; // 2nd scan
605:         return;
606:     }
607:
608:     if(!FloatingDollarLatch && !DecimalControlLatch) {
609:         IRingControl = true;
610:         return;
611:     }
612:
613:     if(FloatingDollarLatch ||
614:         (ZeroSuppressLatch && !(edit_char_flags & SZ_SIGNIFICANT_DIGIT))) {
615:         next.scan = SCAN_3; // Set 3rd scan
616:         next.subscan = SUB_SCAN_MQ; // Set MQ for skid
617:         next.cycle = CYCLE_B; // B Cycle next
618:         next.subcycle = 2; // 3rd scan
619:         return;
620:     }
621:
622:     if(ZeroSuppressLatch && (edit_char_flags & SZ_SIGNIFICANT_DIGIT)) {
623:         IRingControl = true;
624:         return;
625:     }
626:
627:     assert(!FloatingDollarLatch && !ZeroSuppressLatch);
628:     if(b_temp.TestChar(BCD_0)) {
629:         IRingControl = true;
630:         InstructionCheck ->
631:             SetStop("Instruction Check: Edit B WM 0 ZeroSuppress OFF");
632:         return;
633:     }
634:     else {
635:         IRingControl = true;
636:         return;
637:     }
638:
639:     assert(false); // Can't get here
640: } // End B Cycle, subtype 1
641:
642: if(next.cycle == CYCLE_B && next.subcycle == 2) {
643:     *STAR = *B_AR;
644:     Readout(); // Assembly Channel → Read()
645:     b_temp = B_Reg -> Get(); // Assembly Channel
646:     if(SubScanRing -> State() == SUB_SCAN_MQ) { // Read out B Char
647:         next.subscan = SUB_SCAN_E; // Skid cycle?
648:     }
649:     else if(b_temp.TestChar(BCD_SPACE)) { // Set Extension
650:         if(AsteriskFillLatch) { // Set Extension
651:             B_Reg -> Set(BCD_ASTERISK); // Copy IN
652:         }
653:         else if(FloatingDollarLatch) { // Store NOT
654:             B_Reg -> Set(BCD_DOLLAR); // (TO GET WM ON WM)
655:             IRingControl = true; // Stop after $
656:         }
657:         // Else, leave B field char as a blank
658:     }
659:     else if(b_temp.TestChar(BCD_0) || // Stop after $
660:             (b_temp.TestChar(BCD_PERIOD) && DecimalControlLatch) ) {

```

```
661:     if(ZeroSuppressLatch) {
662:         Assembly Channel b_Reg -> Set( AsteriskFillLatch ? BCD_ASTERISK : BCD_SPACE );#
663:         IRingControl = b_temp.TestChar(BCD_PERIOD); // Done if period
664:     }
665:     else if(b_temp.TestChar(BCD_PERIOD)) {
666:         IRingControl = true;
667:     }
668:     // Otherwise, just store the 0
669: } // End 0 or . w decimal control
670: else if(b_temp.TestChar(BCD_PERIOD)) {
671:     assert(!DecimalControlLatch);
672:     // Just store the period back
673: }
674: else {
675:     assert(!b_temp.TestChar(BCD_SPACE) && !b_temp.TestChar(BCD_0) &&
676:           !b_temp.TestChar(BCD_PERIOD));
677:     // Just store other characters back.
678: }
679:
680: Fix: Text Assembly Channel
681: Store( AssemblyChannel -> Select (
682:     TAassemblyChannel::AsmChannelWMNone,
683:     TAassemblyChannel::AsmChannelZonesB,
684:     false,
685:     TAassemblyChannel::AsmChannelSignNone,
686:     TAassemblyChannel::AsmChannelNumB ) );
687: Cycle();
688: if(!IRingControl) {
689:     assert(ScanRing -> State() == SCAN_3); // Regen 3rd scan
690: }
691: return; // Perpahs with IRingControl Set
692: } // End B Cycle, subtype 2
693: } // End Edit Instruction
```

```
1: //-----
2: #ifndef UI1410CPUTH
3: #define UI1410CPUTH
4:
5: #include "ubcd.h"
6:
7: typedef void (_closure *TInstructionExecuteRoutine)();
8:
9: extern long ten_thousands[], thousands[], hundreds[], tens[];
10: extern long scan_mod[];
11: extern unsigned char sign_normalize_table[];
12: extern unsigned char sign_complement_table[];
13: extern unsigned char sign_negative_table[];
14:
15: struct OpCodeCommonLines {
16:     unsigned short ReadOut;
17:     unsigned short Operational;
18:     unsigned short Control;
19: };
20:
21:
22: extern struct OpCodeCommonLines OpCodeTable[64];
23:
24: extern int IndexRegisterLookup [];
25:
26: // Table of execute routines. These have to be closures to inherit
27: // the object pointer (CPU). Closures have to be initialized at runtime.
28: // We do it in the constructor.
29:
30: extern TInstructionExecuteRoutine InstructionExecuteRoutine[64];
31:
32: //
33: // Classes (types) used to implement the emulator, including the
34: // final class defining what is in the CPU, T1410CPU.
35:
36: //
37: // Abstract class designed to build lists of objects affected by Program
38: // Reset and Computer Reset
39: //
40:
41: class TCpuObject : public TObject {
42:
43: public:
44:     TCpuObject();                                // Constructor to init data
45:     virtual void OnComputerReset() = 0;           // Called during Computer Reset
46:     virtual void OnProgramReset() = 0;             // Called during Program Reset
47:
48: protected:
49:     bool DoesProgramReset;                      // true if this is reset by P.R. button
50:
51: public:
52:     TCpuObject *NextReset;
53: };
54:
55:
56: //
57: // A second abstract class of objects that not only react to the Resets,
58: // but also have entries on the display panel.
59: //
60:
61: class TDisplayObject : public TCpuObject {
62:
63: public:
64:     TDisplayObject();                            // Constructor to init data.
65:     virtual void Display() = 0;                  // Called to display this item
66:     virtual void LampTest(bool b) = 0;            // Called to start/end lamp test
```

```
67:
68: public:
69:     TDisplayObject *NextDisplay;
70: };
71:
72: □
73:
74: // Class TDisplayObjects are indicators: They just
75: // display other things. As a result, they need a pointer to
76: // a function returning bool in order to decide what to do.
77:
78: class TDisplayIndicator : public TDisplayObject {
79:
80: protected:
81:     TLabel *lamp;
82:     bool (_closure *display)();
83:
84: public:
85:
86:     // The constructor requires a pointer to a lamp and a pointer to
87:     // a function that can calculate lamp state.
88:
89:     // We use a closure so that we don't have to pass a pointer to the
90:     // stuff the display function needs (an object pointer is automatically
91:     // embedded in an _closure *)
92:
93: TDisplayIndicator(TLabel *l,bool (_closure *func)()) {
94:     lamp = l;
95:     display = func;
96: }
97:
98: virtual void OnComputerReset() { ; }      // These have no state to reset
99: virtual void OnProgramReset() { ; }        // These have no state to reset
100:
101: void Display() {
102:     lamp -> Enabled = display();
103:     lamp -> Repaint();
104: }
105:
106: void LampTest(bool b) {
107:     lamp -> Enabled = (b ? true : display());
108:     lamp -> Repaint();
109: }
110: };
111:
112: □
113:
114: //
115: // Class of TDisplayObjects that are latches:
116: // that can be set, reset and their state read out. Some are
117: // reset by Program Reset (PR) some are not.
118: //
119:
120: class TDisplayLatch : public TDisplayObject {
121:
122: protected:
123:     bool state;                                // Latches can be set or reset
124:     bool doprogramreset;                        // Some are reset by PR, some are not.
125:     TLabel *lamp;                             // Pointer to display lamp.
126:
127: public:
128:     TDisplayLatch(TLabel *l);                  // Constructor - Set up lamp
129:     TDisplayLatch(TLabel *l, bool progresst); // Same, but inhibit PR
130:
131:     virtual void OnComputerReset();           // Define Computer Reset behavior now
132:     virtual void OnProgramReset();            // Define Program Reset behavior now too
```

```
133:
134:     void Display();           // Define display behavior
135:     void LampTest(bool b);   // Define lamp test behavior.
136:
137:     // All you can really do with latches is set/reset/test them
138:
139:     inline void Reset() { state = false; }
140:     inline void Set() { state = true; }
141:     inline void Set(bool b) { state = b; }
142:     void SetStop(char *msg);
143:     inline bool State() { return state; }
144: };
145:
146: □
147:
148: class TRingCounter : public TDisplayObject {
149: private:
150:     char state;           // Override "state" variable !!
151:     char max;             // Max number of entries
152:     TLabel *lastlamp;    // Last lamp to be displayed
153:     TLabel *lastlampCE;  // Last CE lamp to be displayed
154:
155: public:
156:     TLabel **lamps;      // Ptr to array of lamps
157:     TLabel **lampsCE;    // Ptr to array of CE lamps (or 0)
158:
159: public:
160:     TRingCounter(char n); // Construct with # of entries
161:                           // Ring counters are always reset by PR
162:
163:     virtual __fastcall ~TRingCounter(); // Destructor for array of lamps
164:
165:     // Functions inherited from abstract base classes now need definition
166:
167:     void OnComputerReset();
168:     void OnProgramReset();
169:     void Display();
170:     void LampTest(bool b);
171:
172:     // The real meat of the Ring Counter class
173:
174:     inline void Reset() { state = 0; }
175:     inline char Set(char n) { return state = n; }
176:     inline char State() { return state; }
177:     char Next();
178: };
179:
180: □
181:
182: // Data Registers. All are stored as BCD, but we have special
183: // set routines to set or clear special parts for those registers
184: // that don't use all the bits (e.g. the Op register has no WM or
185: // C bits.
186:
187: // Also, a Register can have an optional pointer to an error latch.
188: // If so, then whenever the register is used, a parity check for ODD
189: // parity is made, and if invalid, the error latch is set. (This
190: // includes use during assignment).
191:
192: class TRegister : public TDisplayObject {
193:
194: private:
195:
196:     BCD value;
197:     TLabel *lampER;      // If set, there is an error lamp
198:     TLabel **lamps;       // If set, they point to lamps for WM C B A 8 4 2 1
```

```
199:                         // Any lamp not present MUST be set to 0
200:
201: public:
202:
203:     TRegister() { value = BITC; DoesProgramReset = true; lampER = 0; lamps = 0; }
204:     TRegister(bool b) { value = BITC; DoesProgramReset = b; lampER = 0; lamps = 0; }
205:     TRegister(int i) { value = i; DoesProgramReset = true; lampER = 0; lamps = 0; }
206:     TRegister(int i, bool b) { value = i; DoesProgramReset = b; lampER = 0; lamps =
207:         0; }
208:     inline void OnComputerReset() { Reset(); }
209:     inline void Reset() { value = BITC; }
210:
211:     inline void Set(BCD bcd) { value = bcd; }
212:
213:     inline BCD Get() { return value; }
214:
215:     void operator=(TRegister &source);
216:     void Display();
217:     void LampTest(bool b);
218:
219:     // To set up a register to display, provide a pointer to an error
220:     // lamp (if any), and an array of pointers to data lamps. Data lamps
221:     // for any given bit (e.g. WM) may or may not exist. Order of lamps
222:     // is 1 2 4 8 A B C WM (0 thru 7)
223:
224:     void SetDisplay(TLabel *ler, TLabel **l) {
225:         lampER = ler;
226:         lamps = l;
227:     }
228:
229:     void OnProgramReset() {
230:         if(DoesProgramReset) {
231:             Reset();
232:         }
233:     }
234: };
235: □
236: // Address Registers. For efficiency, we keep both binary and
237: // the real 2-out-of-5 code representations. If either one is
238: // valid, and the other representation is requested, we convert
239: // on the fly (and mark that representation valid).
240:
241: // The Set() function effectively implements the Address Channel.
242:
243: class TAddressRegister : public TCpuObject {
244:
245: private:
246:
247:     long i_value;           // Integer equivalent of register
248:     bool i_valid;          // True if integer rep. is valid
249:     bool set[5];            // True if corresponding digit is set
250:     TWOOF5 digits[5];      // Original 2 out of 5 code representation
251:     bool d_valid;          // True if digit rep. is valid
252:     char *name;
253:
254: public:
255:
256:     void OnComputerReset() { Reset(); };
257:     void OnProgramReset() { };
258:
259:     TAddressRegister();    // Constructor / initialization
260:     bool IsValid();        // Returns true if all digits set
261:     long Gate();            // Returns integer value if valid, -1 if not.
262:     BCD GateBCD(int i);    // Returns a single digit
263:     void Set(TWOOF5 digit,int index); // Sets a digit
```

```
264:     void Set(long value); // Sets whole register from binary (address mod)
265:     void Reset(); // Resets the register to blanks
266:
267:     void operator=(TAddressRegister &source); // Assignment
268:
269: };
270:
271: □
272:
273: // This class defines the A Channel - basically, it defines what
274: // register is selected when the A channel is accessed.
275:
276: class TAChannel : public TDisplayObject {
277:
278: public:
279:
280:     enum AChannelSelect { A_Channel_None = 0, A_Channel_A = 1,
281:                           A_Channel_Mod = 2, A_Channel_E = 3, A_Channel_F = 4 };
282:
283: private:
284:
285:     enum AChannelSelect AChannelSelect;
286:     TLabel *lamps[4];
287:
288: public:
289:
290:     TAChannel(); // Constructor
291:
292:     void OnComputerReset() {
293:         Reset();
294:     }
295:
296:     void OnProgramReset() {
297:         Reset();
298:     }
299:
300:     inline BCD Select(enum AChannelSelect sel); // Select input to A Channel
301:     inline BCD Select(); // Use whatever was last selected
302:
303:     inline void Reset() { AChannelSelect = A_Channel_None; }
304:
305:     void Display();
306:     void LampTest(bool b);
307: };
308:
309: □
310:
311: // The Assembly Channel: The 1410 Mix-Master!
312:
313: class TAssemblyChannel : public TDisplayObject {
314:
315: public:
316:
317:     enum AsmChannelZonesSelect {
318:         AsmChannelZonesNone = 0, AsmChannelZonesB = 1, AsmChannelZonesA = 2
319:     };
320:
321:     enum AsmChannelWMSelect {
322:         AsmChannelWMNone = 0, AsmChannelWMB = 1, AsmChannelWMA = 2
323:     }; AsmChannelWMSet = 4
324:
325:     enum AsmChannelNumericSelect {
326:         AsmChannelNumNone = 0, AsmChannelNumB = 1, AsmChannelNumA = 2,
327:         AsmChannelNumAdder = 3 AsmChannelNumZero = 4
328:     };
329:
```

```

330:     enum AsmChannelSignSelect {
331:         AsmChannelSignNone, AsmChannelSignB = 1, AsmChannelSignA = 2,
332:         AsmChannelSignLatch = 3
333:     };
334:
335: private:
336:
337:     BCD value;           // We remember value, for efficiency
338:     bool valid;          // True if value is set. Reset each cycle!
339:
340:     enum AsmChannelZonesSelect AsmChannelZonesSelect;
341:     enum AsmChannelWMSelect AsmChannelWMSelect;
342:     enum AsmChannelNumericSelect AsmChannelNumericSelect;
343:     enum AsmChannelSignSelect AsmChannelSignSelect;
344:     bool AsmChannelInvertSign;
345:
346:     TLabel *AssmLamps[8];
347:     TLabel *AssmComplLamps[8];
348:     TLabel *AssmERLamp;
349:
350: public:
351:
352:     TAssemblyChannel();
353:
354:     void OnComputerReset() { Reset(); }
355:     void OnProgramReset() { Reset(); }
356:
357:     void Display();
358:     void LampTest(bool b);
359:
360:     BCD Select();           // Uses last value and state
361:
362:     BCD Select(
363:         enum AsmChannelWMSelect WMSelect,
364:         enum AsmChannelZonesSelect ZoneSelect,
365:         bool InvertSign,
366:         enum AsmChannelSignSelect SignSelect,
367:         enum AsmChannelNumericSelect NumSelect
368:     );
369:
370:     BCD Get();
371:
372:     BCD Set(BCD v) { value = v; valid = true; }    ↗ Asm Channel Char Set = true;
373:
374:     void Reset();
375:
376:     BCD GateAChannelToAssembly(BCD v);
377:
378:     BCD GateBRegToAssembly(BCD v);
379:
380: };
381:
382: □
383:
384: // This class defines what is in an I/O Channel
385:
386: #define IOCHNOTREADY 1
387: #define IOCHBUSY 2
388: #define IOCHDATACHECK 4
389: #define IOCHCONDITION 8
390: #define IOCHNOTTRANSFER 16
391: #define IOCHWLRECORD 32
392:
393: #define IOLAMPNOTREADY 0
394: #define IOLAMPBUSY 1
395: #define IOLAMPDATACHECK 2

```

```
396: #define IOLAMPCONDITON 3
397: #define IOLAMPNOTTRANSFER 4
398: #define IOLAMPWLRECORD 5
399:
400: class T1410Channel : public TDisplayObject {
401:
402: public:
403:
404:     // Functions inherited from abstract base class classes now need definition
405:
406:     void OnComputerReset();
407:     void OnProgramReset();
408:     void Display();
409:     void LampTest(bool b);
410:
411: private:
412:
413:     // Channel information
414:
415:     int ChStatus;                                // Channel status (see defines)
416:
417:     TLabel *ChStatusDisplay[6];                  // Channel status lights
418:
419: public:
420:
421:     TRegister *ChOp;
422:     TRegister *ChUnitType;
423:     TRegister *ChUnitNumber;
424:     TRegister *ChR1, *ChR2;
425:
426:     TDisplayLatch *ChInterlock;
427:     TDisplayLatch *ChRBCTInterlock;
428:     TDisplayLatch *ChRead;
429:     TDisplayLatch *ChWrite;
430:     TDisplayLatch *ChOverlap;
431:     TDisplayLatch *ChNotOverlap;
432:
433:     enum TapeDensity {
434:         DENSITY_200_556 = 0, DENSITY_200_800 = 1, DENSITY_556_800 = 2
435:     } TapeDensity;
436:
437:
438:     // Methods
439:
440:     T1410Channel(                                     // Constructor
441:         TLabel *LampInterlock,
442:         TLabel *LampRBCTInterlock,
443:         TLabel *LampRead,
444:         TLabel *LampWWrite,
445:         TLabel *LampOverlap,
446:         TLabel *LampNotOverlap,
447:         TLabel *LampNotRead,
448:         TLabel *LampBusy,
449:         TLabel *LampDataCheck,
450:         TLabel *LampCondition,
451:         TLabel *LampWLRecord,
452:         TLabel *LampNoTransfer
453:     );
454:
455:     inline int SetStatus(int i) { return ChStatus = i; }
456:     inline int GetStatus() { return ChStatus; }
457: };
458:
459:
460: □
461: // This class defines what is actually inside the CPU.
```

```
462:  
463: #define MAXCHANNEL 2  
464: #define CHANNEL1 0  
465: #define CHANNEL2 1  
466:  
467: #define STORAGE 80000  
468:  
469: #define I_RING_OP 0  
470: #define I_RING_1 1  
471: #define I_RING_2 2  
472: #define I_RING_3 3  
473: #define I_RING_4 4  
474: #define I_RING_5 5  
475: #define I_RING_6 6  
476: #define I_RING_7 7  
477: #define I_RING_8 8  
478: #define I_RING_9 9  
479: #define I_RING_10 10  
480: #define I_RING_11 11  
481: #define I_RING_12 12  
482:  
483: #define A_RING_1 0  
484: #define A_RING_2 1  
485: #define A_RING_3 2  
486: #define A_RING_4 3  
487: #define A_RING_5 4  
488: #define A_RING_6 5  
489:  
490: #define CLOCK_A 0  
491: #define CLOCK_B 1  
492: #define CLOCK_C 2  
493: #define CLOCK_D 3  
494: #define CLOCK_E 4  
495: #define CLOCK_F 5  
496: #define CLOCK_G 6  
497: #define CLOCK_H 7  
498: #define CLOCK_J 8  
499: #define CLOCK_K 9  
500:  
501: #define SCAN_N 0  
502: #define SCAN_1 1  
503: #define SCAN_2 2  
504: #define SCAN_3 3  
505:  
506: #define SUB_SCAN_NONE 0  
507: #define SUB_SCAN_U 1  
508: #define SUB_SCAN_B 2  
509: #define SUB_SCAN_E 3  
510: #define SUB_SCAN_MQ 4  
511:  
512: #define CYCLE_A 0  
513: #define CYCLE_B 1  
514: #define CYCLE_C 2  
515: #define CYCLE_D 3  
516: #define CYCLE_E 4  
517: #define CYCLE_F 5  
518: #define CYCLE_I 6  
519: #define CYCLE_X 7  
520:  
521: class T1410CPU {  
522:  
523: private:  
524:  
525:     BCD core[STORAGE];  
526:  
527: public:
```

```

528: // Wiring list
529: TCpuObject *ResetList; // List of latches.
530: TDisplayObject *DisplayList; // List of displayable things
531:
532:
533:
534: // Data Registers
535:
536: TRegister *A_Reg, *B_Reg, *Op_Reg, *Op_Mod_Reg;
537:
538: // Address Registers
539:
540: TAddressRegister *STAR; // Storage Address Register
541: // AKA MAR (Memory Address Register)
542:
543: TAddressRegister *A_AR, *B_AR, *C_AR, *D_AR, *E_AR, *F_AR;
544:
545: TAddressRegister *I_AR; // Instruction Counter
546:
547:
548: // Channels
549:
550: T1410Channel *Channel[MAXCHANNEL]; // 2 I/O Channels.
551:
552: TAChannel *AChannel; // A Channel
553: TAssemblyChannel *AssemblyChannel; // Assembly Channel
554:
555: // Indicators
556:
557: TDisplayIndicator *OffNormal; // OFF NORMAL Indicator
558:
559: // Ring Counters
560:
561: TRingCounter *IRing; // Instruction decode ring
562: TRingCounter *ARing; // Address decode ring
563: TRingCounter *ClockRing; // Cycle Clock
564: TRingCounter *ScanRing; // Address Modification Mode
565: TRingCounter *SubScanRing; // Arithmetic Scan type
566: TRingCounter *CycleRing; // CPU Cycle type
567:
568: // Latches with Indicators
569:
570: TDisplayLatch *CarryIn; // Carry latch
571: TDisplayLatch *CarryOut; // Adder has generated carry
572: TDisplayLatch *AComplement; // A channel complement
573: TDisplayLatch *BComplement; // B channel complement
574: TDisplayLatch *CompareBGTA; // B > A
575: TDisplayLatch *CompareBEQA; // B = A
576: TDisplayLatch *CompareBLTA; // B < A NOTE: On after C. Reset.
577: TDisplayLatch *Overflow; // Arithmetic Overflow
578: TDisplayLatch *DivideOverflow; // Divide Overflow
579: TDisplayLatch *ZeroBalance; // Zero arithmetic result
580:
581: // Check Latches
582:
583: TDisplayLatch *AChannelCheck; // A Channel parity error
584: TDisplayLatch *BChannelCheck; // B Channel parity error
585: TDisplayLatch *AssemblyChannelCheck; // Assembly Channel parity error
586: TDisplayLatch *AddressChannelCheck; // Address Channel parity error
587: TDisplayLatch *AddressExitCheck; // Validity error at address reg.
588: TDisplayLatch *ARegisterSetCheck; // A register failed to reset
589: TDisplayLatch *BRegisterSetCheck; // B register failed to reset
590: TDisplayLatch *OpRegisterSetCheck; // Op register failed to set
591: TDisplayLatch *OpModifierSetCheck; // Op modifier failed to set
592: TDisplayLatch *ACharacterSelectCheck; // Incorrect A channel gating
593: TDisplayLatch *BCharacterSelectCheck; // Incorrect B channel getting

```

```
594:     TDisplayLatch *IOInterlockCheck;           // Program did not check I/O
595:     TDisplayLatch *AddressCheck;             // Program gave bad address
596:     TDisplayLatch *RBCInterlockCheck;        // Program did not check RBC
597:     TDisplayLatch *InstructionCheck;         // Program issued invalide op
598:
599:
600:     // Switches
601:
602:     enum Mode {                                // Mode switch, values must match
603:         MODE_RUN = 0, MODE_DISPLAY = 1, MODE_ALTER = 2,
604:         MODE_CE = 3, MODE_IE = 4, MODE_ADDR = 5
605:     } Mode;
606:
607:     enum AddressEntry {
608:         ADDR_ENTRY_I = 0, ADDR_ENTRY_A = 1, ADDR_ENTRY_B = 2,
609:         ADDR_ENTRY_C = 3, ADDR_ENTRY_D = 4, ADDR_ENTRY_E = 5,
610:         ADDR_ENTRY_F = 6
611:     } AddressEntry;
612:
613:     // The entry below is for the state of the 1415 Storage Scan SWITCH
614:     // (The storage scan modification mode is in the Ring ScanRing)
615:
616:     enum StorageScan {
617:         SSCAN_OFF = 0, SSCAN_LOAD_1 = 1, SSCAN_LOAD_0 = 2,
618:         SSCAN_REGEN_0 = 3, SSCAN_REGEN_1 = 4
619:     } StorageScan;
620:
621:     enum CycleControl {
622:         CYCLE_OFF = 0, CYCLE_LOGIC = 1, CYCLE_STORAGE = 2
623:     } CycleControl;
624:
625:     enum CheckControl {
626:         CHECK_STOP = 0, CHECK_RESTART = 1, CHECK_RESET = 2
627:     } CheckControl;
628:
629:     bool DiskWrInhibit;
630:     bool AsteriskInsert;
631:     bool InhibitPrintOut;
632:
633:     BCD BitSwitches;
634:
635:     // CPU state latches
636:
637:     bool StopLatch;                           // True to stop CPU
638:     bool StopKeyLatch;                      // True if STOP button pressed
639:     bool DisplayModeLatch;                  // True if we are displaying storage
640:     bool ProcessRoutineLatch;               // True if we are in Run or IE mode
641:     bool BranchLatch;                      // True if we are to branch
642:     bool BranchTo1Latch;                   // True to branch to 1
643:     bool LastInstructionReadout;          // True at end of Instruction fetch
644:     bool IRingControl;                     // Set to start Instruction Fetch
645:
646:     bool IOMoveModeLatch;                  // I/O Latches
647:     bool IOLoadModeLatch;                 // to check for I/O Overlap
648:
649:     bool StorageWrapLatch;
650:
651:     int IOChannelSelect;                  // One if if channel 2 op.
652:
653:     // Index latches are rolled up into an int. The BA zones from the
654:     // tens and hundreds positions are shifted and the 4 bits together
655:     // generate a value from 0 to 15.
656:
657:     int IndexLatches;
658:
659:     // Methods
```

```
660:
661:     T1410CPU();                                // Constructor
662:     void Display();                           // Run thru the display list
663:     void Cycle();                            // Used for common CPU Cycles
664:
665:     // The 1410 Adder accepts BCD inputs, a Carry Latch and complement
666:     // flags and returns a sum in BCD, possible setting Carry Out.
667:
668:     BCD Adder(BCD A,int Complement_A,BCD B,int Complement_B);
669:     BCD AdderResult;                         // For the Assembly Channel
670:
671: private:
672:
673:     // Indicator Routines
674:     // Normally, these will be accessed via a bool (__closure *func)()
675:
676:     bool IndicatorOffNormal();
677:
678: public:
679:
680:     // Core operations (using STAR/MAR and B Data Register)
681:
682:     void Readout();                          // Reads out one storage character
683:     void Store();                           // Stores character in B Data Register
684:     void Store(BCD bcd);                   // Sets B Data Register, then stores
685:     void SetScan(char s);                  // Sets Scan Modification value
686:
687:     long STARSscan();                     // Applies Scan CTRL modification to STAR,
688:                                         // and returns the results - suitable to assign
689:
690:     long STARMOD(int mod);                // Similar, but provides direct +1/0/-1 mod
691:
692:     void LoadCore(char *file);           // Loads core from a file
693:     void DumpCore(char *file);          // Dumps core to a file
694:
695: public:
696:
697:     // Instruction operations
698:
699:     unsigned short OpReadOutLines;
700:     unsigned short OpOperationalLines;
701:     unsigned short OpControlLines;
702:
703:     void DoStartClick();                 // START pressed (moved from UI1410PWR)
704:     void InstructionDecodeStart();    // Starts instruction decode processing
705:     void InstructionDecode();        // Remainder of instruction decode
706:     void InstructionDecodeIARAdvance(); // Conditionally advance IAR
707:
708:     void InstructionIndexStart();    // Starts up indexing operation
709:     void InstructionIndex();         // Does the actual indexing
710:
711:     void InstructionExecuteInvalid(); // Default instruction execute routine
712:
713:     void InstructionArith();         // Add and Subtract
714:     void InstructionZeroArith();    // Zero and Add, Zero and Subtract
715:
716: };
717:
718: extern T1410CPU *CPU;
719:
720: //-----
721: #endif
722:
```

## UI1410CPU.cpp

```
1: // This Unit provides the functionality behind some of my private
2: // types for the 1410 emulator
3:
4: //-----
5: #include <vcl\vcl.h>
6: #include <assert.h>
7: #include <stdlib.h>
8: #pragma hdrstop
9:
10: #include "UI1410CPU.h"
11: #include "UI1415L.h"
12: #include "UI1415CE.h"
13: #include "UI1410PWR.h"
14: #include "UI1410DEBUG.h"
15: #include "UI1410CPU.h"
16: #include "UI1410INST.H"
17:
18: //-----
19:
20: // Declarations for Borland VCL controls
21:
22: #include <vcl\Classes.hpp>
23: #include <vcl\Controls.hpp>
24: #include <vcl\StdCtrls.hpp>
25:
26: // Table of execute routines. These have to be closures to inherit
27: // the object pointer (CPU). Closures have to be initialized at runtime.
28: // We do it in the constructor.
29:
30: TInstructionExecuteRoutine InstructionExecuteRoutine[64];
31:
32:
33: // Core pre-load (gives this virtual 1410 a 7010-like load capability
34: // Just hit Computer Reset then Start to boot from tape
35:
36: char *core_load_chars = "AL%BO00012$N..";
37: char core_load_wm[] = { 0,1,0,0,0,0,0,0,0,0,1,1,1 };
38:
39: // Tables which pre-multiply integers by decimal numbers, for efficiency.
40:
41: long ten_thousands[] = {
42:     0,10000,20000,30000,40000,50000,60000,70000,80000,90000
43: };
44:
45: long thousands[] = {
46:     0,1000,2000,3000,4000,5000,6000,7000,8000,9000
47: };
48:
49: long hundreds[] = {
50:     0,100,200,300,400,500,600,700,800,900
51: };
52:
53: long tens[] = { 0,10,20,30,40,50,60,70,80,90 };
54:
55: // Storage Scan modification values
56: // Correspond to the values in the enum StorageScan
57:
58: long scan_mod[] = { 0, -1, +1, -1 };
59:
60: // Tables for processing signs (shifted right 4 bits!)
61:
62: // Table to take a sign bit configuration (BA) and normalize it.
63: // Plus sign is normalized to BA, Minus sign is normalized to just A
64:
65: unsigned char sign_normalize_table[] = { 3, 3, 2, 3 };
66:
```

```
67: // Same thing, but inverts the sign in the process
68:
69: unsigned char sign_complement_table[] = { 2, 2, 3, 2 };
70:
71: // Same thing, but indicates negative by 1
72:
73: unsigned char sign_negative_table[] = { 0, 0, 1, 0 };
74:
75: // Implementation of TCpuObject (Abstract Base Class)
76:
77: // Everything in the CPU is on the reset list.
78: // Everything is reset by Computer Reset
79: // NOT Everything is reset by Program Reset!
80:
81: TCpuObject::TCpuObject()
82: {
83:     NextReset = CPU -> ResetList;
84:     CPU -> ResetList = this;
85: }
86:
87: // Implementation of TDisplayObject (Abstract Base Class)
88:
89: // What is special about these objects is that they are on the display list.
90:
91: TDisplayObject::TDisplayObject()
92: {
93:     NextDisplay = CPU -> DisplayList;
94:     CPU -> DisplayList = this;
95: }
96:
97: □
98:
99: // Implementation of TDisplayLatch (Display Latch Base Class)
100:
101: // The constructor initializes the latch and sets the pointer to a lamp.
102: // A pointer to a lamp is required.
103:
104: // Default is to be reset by a Program Reset, but this can be overridden
105: // when the constructor is called, if necessary.
106:
107: TDisplayLatch::TDisplayLatch(TLabel *l)
108: {
109:     state = false;
110:     doprogramreset = true;
111:     lamp = l;
112: }
113:
114: // The second constructor does the same thing, but is passed a variable
115: // which indicates whether or not the latch should be reset by Program
116: // Reset.
117:
118: TDisplayLatch::TDisplayLatch(TLabel *l, bool progreset)
119: {
120:     state = false;
121:     doprogramreset = progreset;
122:     lamp = l;
123: }
124:
125: // All Displayable Latches are reset on a COMPUTER RESET
126:
127: void TDisplayLatch::OnComputerReset()
128: {
129:     Reset();
130: }
131:
132: // Whether or not a displayable latch is reset by program reset depends
```

```
133: //  on the latch.
134:
135: void TDisplayLatch::OnProgramReset()
136: {
137:     if(doprogramreset) {
138:         Reset();
139:     }
140: }
141:
142: // Routine to set a latch and set the CPU Stop Latch at the same time
143: // Typically used for error latches.
144:
145: void TDisplayLatch::SetStop(char *msg) {
146:     state = true;
147:     CPU -> StopLatch = true;
148:     if(msg != 0) {
149:         F1410Debug -> DebugOut(msg);
150:     }
151: }
152:
153: // When the Display routine is called, it sets or resets the lamp,
154: // depending on the current state.
155:
156: void TDisplayLatch::Display()
157: {
158:     lamp -> Enabled = state;
159:     lamp -> Repaint();
160: }
161:
162: // On a lamp test, light all the lamp.
163: // On reset of a lamp test, display the current state.
164:
165: void TDisplayLatch::LampTest(bool b)
166: {
167:     lamp -> Enabled = (b ? true : state);
168:     lamp -> Repaint();
169: }
170:
171: □
172:
173: // Implementation of TRingCounter (Ring Counter Class)
174:
175: // The constructor sets the max state of the ring, and resets the ring
176: // It also allocates an array of pointers to the lamps. The creator must
177: // fill in that array, however. Initially, the lamp pointers are empty,
178: // (which means no lamp is attached to that state).
179:
180: TRingCounter::TRingCounter(char n)
181: {
182:     int i;
183:
184:     state = 0;
185:     max = n;
186:     lastlamp = 0;
187:     lastlampCE = 0;
188:     lampsCE = 0;
189:
190:     lamps = new TLabel*[n];
191:     for(i=0; i < n; ++i) {
192:         lamps[i] = 0;
193:     }
194: }
195:
196: // The destructor frees up the array of lamps. Probably will never use.
197:
198: __fastcall TRingCounter::~TRingCounter()
```

```
199: {
200:     delete[] lamps;
201:     if(lampsCE != 0) {
202:         delete[] lampsCE;
203:     }
204: }
205:
206: // All Ring counters are reset on PROGRAM and COMPUTER RESET
207:
208: void TRingCounter::OnComputerReset()
209: {
210:     state = 0;
211: }
212:
213: void TRingCounter::OnProgramReset()
214: {
215:     state = 0;
216: }
217:
218: // When the Display routine is called, it resets the lamp corresponding
219: // to the state when it last displayed, and then displays the current state
220: // If there is no lamp associated with a state (lamp pointer is null),
221: // then don't display any lamp.
222:
223: void TRingCounter::Display()
224: {
225:     if(lastlamp) {
226:         lastlamp -> Enabled = false;
227:         lastlamp -> Repaint();
228:     }
229:     if(lamps[state] == 0) {
230:         lastlamp = 0;
231:         return;
232:     }
233:     lamps[state] -> Enabled = true;
234:     lamps[state] -> Repaint();
235:     lastlamp = lamps[state];
236:
237:     if(lampsCE != 0) {
238:         if(lastlampCE) {
239:             lastlampCE -> Enabled = false;
240:             lastlampCE -> Repaint();
241:         }
242:         if(lampsCE[state] == 0) {
243:             lastlampCE = 0;
244:             return;
245:         }
246:         lampsCE[state] -> Enabled = true;
247:         lampsCE[state] -> Repaint();
248:         lastlampCE = lampsCE[state];
249:     }
250: }
251:
252: // On a lamp test, light all the associated lamps.
253: // On reset of a lamp test, clear them all, then display the current state.
254:
255: void TRingCounter::LampTest(bool b)
256: {
257:     int i;
258:
259:     for(i = 0; i < max; ++i) {
260:         if(lamps[i] != 0) {
261:             lamps[i] -> Enabled = b;
262:             lamps[i] -> Repaint();
263:         }
264:         if(lampsCE != 0 && lampsCE[i] != 0) {
```

```
265:             lampsCE[i] -> Enabled = b;
266:             lampsCE[i] -> Repaint();
267:         }
268:     }
269:     if(!b) {
270:         lastlamp = 0;
271:         lastlampCE = 0;
272:         this -> Display();
273:     }
274: }
275:
276: // Next advances to the next state (or back to the start if appropriate)
277: // Returns current state. This is only useful for true Ring counters.
278:
279: char TRingCounter::Next()
280: {
281:     if(++state >= max) {
282:         state = 0;
283:     }
284:     return(state);
285: }
286:
287: □
288:
289: // Implementation of Simple Registers
290:
291: // Assignment: Just assign the value, NOT the things from TCpuObject!!
292: // (Those need to stay unchanged as they should be invariant once the
293: // register is created: it's position on the reset list and whether or
294: // not it is affected by Program Reset, for example.
295:
296: void TRegister::operator=(TRegister &source)
297: {
298:     value = source.value;
299: }
300:
301: // Display and LampTest. These only do anything if the display variables
302: // are actually set.
303:
304: void TRegister::Display()
305: {
306:     int bitmask = 0x1;
307:     int i;
308:
309:     for(i=0; i < 8; ++i) {
310:         if(lamps != 0 && lamps[i] != 0) {
311:             lamps[i] -> Enabled = ((value.ToInt() & bitmask) != 0);
312:             lamps[i] -> Repaint();
313:         }
314:         bitmask <= 1;
315:     }
316:
317:     if(lampER != 0) {
318:         lampER -> Enabled = (!value.CheckParity());
319:         lampER -> Repaint();
320:     }
321: }
322:
323: void TRegister::LampTest(bool b)
324: {
325:     int i;
326:
327:     if(b) {
328:         for(i=0; i < 8; ++i) {
329:             if(lamps != 0 && lamps[i] != 0) {
330:                 lamps[i] -> Enabled = true;
```

```
331:             lamps[i] -> Repaint();
332:         }
333:     }
334:     if(lampER != 0) {
335:         lampER -> Enabled = true;
336:         lampER -> Repaint();
337:     }
338: }
339: else {
340:     Display();
341: }
342: }
343:
344: □
345:
346: // Implementation of Address Registers
347:
348: // The constructor just initializes things so that we know that the
349: // address register contains an invalid value.
350:
351: TAddressRegister::TAddressRegister()
352: {
353:     i_valid = false;
354:     i_value = 0;
355:     d_valid = false;
356:     set[0] = set[1] = set[2] = set[3] = set[4] = false;
357:     DoesProgramReset = false;
358: }
359:
360: // A function to determine whether or not the address register contains
361: // a valid value.
362:
363: bool TAddressRegister::IsValid()
364: {
365:     if(i_valid || d_valid ||
366:         !(set[0] && set[1] && set[2] && set[3] && set[4])) {
367:         return(true);
368:     }
369:     return(false);
370: }
371:
372: // A routine to reset an address register: to binary 0. Note that this
373: // means it is invalid (i.e. will cause an Address Exit Check if you try
374: // and actually use the value except to print it out on the console.
375:
376: void TAddressRegister::Reset()
377: {
378:     TWOOF5 zero;
379:
380:     i_valid = false;
381:     d_valid = false;
382:     set[0] = set[1] = set[2] = set[3] = set[4] = false;
383:     digits[0] = digits[1] = digits[2] = digits[3] = digits[4] = zero;
384:     i_value = 0;
385: }
386:
387: // A routine to get the value of an address register. Note that if
388: // the value is invalid, the result is an Address Exit Check.
389:
390: long TAddressRegister::Gate()
391: {
392:     if(i_valid) {
393:         return(i_value);
394:     }
395:     else if(IsValid()) {
396:         i_value = ten_thousands[digits[0].ToInt()] +
```

```
397:             thousands[digits[1].ToInt()] +
398:             hundreds[digits[2].ToInt()] +
399:             tens[digits[3].ToInt()] + digits[4].ToInt();
400:         i_valid = true;
401:         d_valid = true;
402:         return(i_value);
403:     }
404: else {
405:     CPU -> AddressExitCheck ->
406:         SetStop("Address Exit Check during Gate()");
407:     return(-1);
408: }
409: }
410:
411: // Get a single character from an address register. This is valid even
412: // if the value in the register is invalid.
413:
414: BCD TAddressRegister::GateBCD(int i)
415: {
416:     if(d_valid) {
417:         if(digits[i-1].ToInt() < 0) {
418:             CPU -> AddressExitCheck ->
419:                 SetStop("Address Exit Check during GateBCD(#1)");
420:             F1415L -> DisplayAddrChannel(digits[i-1],true);
421:             return(0);
422:         }
423:         return(digits[i-1].ToBCD());
424:     }
425:     else if(i_valid) {
426:         digits[4] = i_value % 10;
427:         digits[3] = (i_value % 100)/10;
428:         digits[2] = (i_value % 1000)/100;
429:         digits[1] = (i_value % 10000)/1000;
430:         digits[0] = i_value /10000;
431:         d.valid = true;
432:         set[0] = set[1] = set[2] = set[3] = set[4] = true;
433:         if(digits[i-1].ToInt() < 0) {
434:             CPU -> AddressExitCheck ->
435:                 SetStop("Address Exit Check during GateBCD(#2)");
436:             return(0);
437:         }
438:         return(digits[i-1].ToBCD());
439:     }
440:     else {
441:         // Register is not set, but it is still valid to read out a single
442:         // digit. This should not cause an error.
443:
444: /*     CPU -> AddressExitCheck ->
445:         SetStop("Address Exit Check during GateBCD(Contents not valid)");
446:         F1415L -> DisplayAddrChannel(TWOOF5(0),true);
447:         return(BCD(0));
448: */
449: */
450:
451:     if(set[i-1] && digits[i-1].ToInt() >= 0) {
452:         return(digits[i-1].ToBCD());
453:     }
454:     else {
455:         return(0);
456:     }
457: }
458: }
459:
460: // A routine to set a single digit of an address register.
461: // This effectively implements the address channel.
462:
```

```
463: void TAddressRegister::Set(TWOOF5 digit,int i)
464: {
465:     if(digit.ToInt() == -1) {
466:         CPU -> AddressChannelCheck ->
467:             SetStop("Address Channel Check while setting address register");
468:         F1415L -> DisplayAddrChannel(digit,true);
469:     }
470:
471:     digits[i-1] = digit;
472:     set[i-1] = true;
473:     if(i == 5 && set[0] && set[1] && set[2] && set[3]) {
474:         d_valid = true;
475:     }
476: }
477:
478: // A routine to set an address register from a binary value. Should
479: // really only be used to reset IAR to 1 during a reset operation.
480:
481: void TAddressRegister::Set(long i)
482: {
483:     d_valid = false;
484:     set[0] = set[1] = set[2] = set[3] = set[4] = false;
485:     i_valid = true;
486:     i_value = i;
487: }
488:
489: // Assignment: Just assign the value, NOT the things from TCpuObject!!
490: // (Those need to stay unchanged as they should be invariant once the
491: // register is created: it's position on the reset list and whether or
492: // not it is affected by Program Reset, for example.
493:
494: // Note that an attempt to assign from an invalid register does do the
495: // set, but also sets the Address Exit Check error in the CPU
496:
497: void TAddressRegister::operator=(TAddressRegister &source)
498: {
499:     int i;
500:
501:     i_value = source.i_value;
502:     i_valid = source.i_valid;
503:     d_valid = source.d_valid;
504:     if(!source.IsValid()) {
505:         CPU -> AddressExitCheck ->
506:             SetStop("Address Exit Check assigning - from register invalid");
507:     }
508:     for(i=0; i < 5; ++i) {
509:         digits[i] = source.digits[i];
510:         set[i] = source.set[i];
511:     }
512: }
513:
514: □
515:
516: // Implementation of A Channel
517:
518: // Constructor. There is only 1 a channel, so it sets pointers to lamps
519: // as well as initializing
520:
521: TAChannel::TAChannel()
522: {
523:     AChannelSelect = A_Channel_None;
524:
525:     lamps[0] = F1415L -> Light_CE_ACh_A;
526:     lamps[1] = F1415L -> Light_CE_ACh_d;
527:     lamps[2] = F1415L -> Light_CE_ACh_E;
528:     lamps[3] = F1415L -> Light_CE_ACh_F;
```

```
529: }
530:
531: // Routine to return what is currently selected on A Channel
532:
533: BCD TACChannel::Select()
534: {
535:     return Select(AChannelSelect);
536: }
537:
538: // Routine to set what the A Channel will select. It also returns that
539: // selected value, for convenience.
540:
541: BCD TACChannel::Select(enum AChannelSelect sel)
542: {
543:     AChannelSelect = sel;
544:
545:     switch(AChannelSelect) {
546:
547:         case A_Channel_None:
548:             return(0);
549:         case A_Channel_A:
550:             return(CPU -> A_Reg -> Get());
551:         case A_Channel_Mod:
552:             return(CPU -> Op_Mod_Reg -> Get());
553:         case A_Channel_E:
554:             return(CPU -> Channel[CHANNEL1] -> ChR2 -> Get());
555:         case A_Channel_F:
556:             if(MAXCHANNEL > 1) {
557:                 return(CPU -> Channel[CHANNEL2] -> ChR2 -> Get());
558:             }
559:             else {
560:                 CPU -> ACharacterSelectCheck ->
561:                     SetStop("A Character Select Check: Channel 2, only 1 configured");
562:                 return(0);
563:             }
564:         default:
565:             CPU -> ACharacterSelectCheck ->
566:                 SetStop("A Character Select Check: Invalid selection");
567:             return(0);
568:     }
569: }
570:
571: // Routine to display the current A Channel Select status
572:
573: void TACChannel::Display()
574: {
575:     int i;
576:
577:     for(i=0; i < 4; ++i) {
578:         lamps[i] -> Enabled = false;
579:     }
580:
581:     switch(AChannelSelect) {
582:
583:         case A_Channel_None:
584:             break;
585:         case A_Channel_A:
586:             lamps[0] -> Enabled = true;
587:             break;
588:         case A_Channel_Mod:
589:             lamps[1] -> Enabled = true;
590:             break;
591:         case A_Channel_E:
592:             lamps[2] -> Enabled = true;
593:             break;
594:         case A_Channel_F:
```

```
595:         if(MAXCHANNEL > 1) {
596:             lamps[3] -> Enabled = true;
597:         }
598:         break;
599:     }
600:
601:     for(i = 0; i < 4; ++i) {
602:         lamps[i] -> Repaint();
603:     }
604: }
605:
606: // Lamp Test
607:
608: void TAChannel::LampTest(bool b)
609: {
610:     int i;
611:
612:     if(b) {
613:         for(i = 0; i < 4; ++i) {
614:             lamps[i] -> Enabled = true;
615:             lamps[i] -> Repaint();
616:         }
617:     }
618:     else {
619:         Display();
620:     }
621: }
622:
623: □
624:
625: // Implementation of Assembly Channel
626:
627: // Private tables for assembly channel. These are bitmasks. In general,
628: // these arrays have elements in the order of their respective enums.
629: // Each element has 3 entries, one byte each for the A Channel, the B Channel/
630: // Register and the Adder
631:
632: struct AssmMask {
633:     int AChannelMask;
634:     int BChannelMask;
635:     int AdderMask;
636: };
637:
638: static struct AssmMask AssmWMMask[] = {
639:     { 0x00, 0x00, 0x00 },           // No WM
640:     { 0x00, 0x80, 0x00 },           // B WM
641:     { 0x80, 0x00, 0x00 }            // A WM
642: };
643:
644: static struct AssmMask AssmZonesMask[] = {
645:     { 0x00, 0x00, 0x00 },           // No Zones
646:     { 0x00, 0x30, 0x00 },           // B Zones
647:     { 0x30, 0x00, 0x00 }            // A Zones
648: };
649:
650: static struct AssmMask AssmNumMask[] = {
651:     { 0x00, 0x00, 0x00 },           // No Numerics
652:     { 0x00, 0x0f, 0x00 },           // B Numerics
653:     { 0x0f, 0x00, 0x00 },           // A Numerics
654:     { 0x00, 0x00, 0x0f }            // Adder Numerics
655: };
656:
657: static struct AssmMask AssmSignMask[] = {
658:     { 0x00, 0x00, 0x00 },           // No Sign
659:     { 0x00, 0x30, 0x00 },           // B Sign
660:     { 0x30, 0x00, 0x00 }            // A Sign
```

```
661:     { 0x00, 0x00, 0x30 }           // Sign Latch
662: };
663:
664: // Constructor
665:
666: TAssemblyChannel::TAssemblyChannel()
667: {
668:     AssmLamps[0] = F1415L -> Light_CE_Assm_1;
669:     AssmLamps[1] = F1415L -> Light_CE_Assm_2;
670:     AssmLamps[2] = F1415L -> Light_CE_Assm_4;
671:     AssmLamps[3] = F1415L -> Light_CE_Assm_8;
672:     AssmLamps[4] = F1415L -> Light_CE_Assm_A;
673:     AssmLamps[5] = F1415L -> Light_CE_Assm_B;
674:     AssmLamps[6] = F1415L -> Light_CE_Assm_C;
675:     AssmLamps[7] = F1415L -> Light_CE_Assm_WM;
676:
677:     AssmComplLamps[0] = F1415L -> Light_CE_Assm_N1;
678:     AssmComplLamps[1] = F1415L -> Light_CE_Assm_N2;
679:     AssmComplLamps[2] = F1415L -> Light_CE_Assm_N4;
680:     AssmComplLamps[3] = F1415L -> Light_CE_Assm_N8;
681:     AssmComplLamps[4] = F1415L -> Light_CE_Assm_NA;
682:     AssmComplLamps[5] = F1415L -> Light_CE_Assm_NB;
683:     AssmComplLamps[6] = F1415L -> Light_CE_Assm_NC;
684:     AssmComplLamps[7] = F1415L -> Light_CE_Assm_NWM;
685:
686:     AssmERLamp = F1415L -> Light_CE_Assm_ER;
687:
688:     Reset();
689: }
690:
691: // Display and LampTest have to display both positive and complement.
692: // Otherwise, very similar to a TRegister
693:
694: void TAssemblyChannel::Display()
695: {
696:     int bitmask = 0x1;
697:     int i;
698:
699:     for(i=0; i < 8; ++i) {
700:         AssmLamps[i] -> Enabled = ((value.ToInt() & bitmask) != 0);
701:         AssmComplLamps[i] -> Enabled = !AssmLamps[i] -> Enabled;
702:         AssmLamps[i] -> Repaint();
703:         AssmComplLamps[i] -> Repaint();
704:         bitmask <= 1;
705:     }
706:
707:     AssmERLamp -> Enabled = (!valid || !value.CheckParity());
708:     AssmERLamp -> Repaint();
709: }
710:
711: void TAssemblyChannel::LampTest(bool b)
712: {
713:     int i;
714:
715:     if(b) {
716:         for(i=0; i < 8; ++i) {
717:             AssmLamps[i] -> Enabled = true;
718:             AssmComplLamps[i] -> Enabled = true;
719:             AssmLamps[i] -> Repaint();
720:             AssmComplLamps[i] -> Repaint();
721:         }
722:         AssmERLamp -> Enabled = true;
723:         AssmERLamp -> Repaint();
724:     }
725:     else {
726:         Display();
```

```

727:     }
728: }
729:
730: // Select sets the Assembly Channel selection variables and then
731: // returns the result
732:
733: BCD TAssemblyChannel::Select(
734:     enum AsmChannelWMSelect WMSelect,
735:     enum AsmChannelZonesSelect ZoneSelect,
736:     bool InvertSign,
737:     enum AsmChannelSignSelect SignSelect,
738:     enum AsmChannelNumericSelect NumSelect)
739: {
740:     bool SignLatch = false; // Ditto for sign latch
741:
742:     BCD Zones, WM, Numerics;
743:
744:     // Remember the selection, for next time
745:
746:     AsmChannelWMSelect = WMSelect;
747:     AsmChannelZonesSelect = ZoneSelect;
748:     AsmChannelInvertSign = InvertSign;
749:     AsmChannelSignSelect = SignSelect;
750:     AsmChannelNumericSelect = NumSelect;
751:
752:     // Can't select both sign and zones!
753:
754:     if(AsmChannelZonesSelect != AsmChannelZonesNone &&
755:         AsmChannelSignSelect != AsmChannelSignNone) {
756:         CPU -> AssemblyChannelCheck ->
757:             SetStop("Assembly Channel Check: Selected both Sign and Zones");
758:         return(0);
759:     }
760:
761:     WM =
762:         (CPU -> AChannel -> Select() & AssmWMMask[AsmChannelWMSelect].AChannelMask)
763:         |
764:         (CPU -> B_Reg -> Get() & AssmWMMask[AsmChannelWMSelect].BChannelMask);
765:
766:     Numerics =
767:         (CPU -> AChannel -> Select() &
768:          AssmNumMask[AsmChannelNumericSelect].AChannelMask) |
769:         (CPU -> B_Reg -> Get() & AssmNumMask[AsmChannelNumericSelect].BChannelMask)
770:         |
771:         (CPU -> AdderResult & AssmNumMask[AsmChannelNumericSelect].AdderMask);
772:
773:     if(AsmChannelSignSelect != AsmChannelSignNone) {
774:         if(AsmChannelSignSelect == AsmChannelSignLatch) {
775:             Zones = (SignLatch ? 0x20 : 0x30);
776:         }
777:         else {
778:             Zones =
779:                 (CPU -> AChannel -> Select() &
780:                  AssmSignMask[AsmChannelSignSelect].AChannelMask) |
781:                  (CPU -> B_Reg -> Get() &
782:                   AssmSignMask[AsmChannelSignSelect].BChannelMask);
783:             }
784:             Zones = Zones >> 4; // Prepare to normalize
785:             Zones = BCD( (AsmChannelInvertSign ?
786:                           sign_complement_table[Zones.ToInt()] :
787:                           sign_normalize_table[Zones.ToInt()]) << 4 );

```

move  
To CPU

```
787:         (CPU -> B_Reg -> Get() &
788:          AssmZonesMask[AsmChannelZonesSelect].BChannelMask);
789:
790:     value = WM | Zones | Numerics;
791:     value.SetOddParity();
792:     valid = true;
793:     return(value);
794: }
795:
796: // Same thing, only we use the existing values. If the value is already
797: // valid, we can take a shortcut.
798:
799: BCD TAssemblyChannel::Select()
800: {
801:     if(valid) {
802:         return(value);
803:     }
804:
805:     return(Select(AsmChannelWMSelect, AsmChannelZonesSelect,
806:                   AsmChannelInvertSign, AsmChannelSignSelect, AsmChannelNumericSelect));
807: }
808:
809: // Get really does the same thing as select, except that it flags an
810: // error if there is not a valid value already.
811:
812: BCD TAssemblyChannel::Get()
813: {
814:     if(valid) {
815:         return(value);
816:     }
817:
818:     CPU -> AssemblyChannelCheck ->
819:         SetStop("Assembly Channel Check: Value not valid at this time");
820:     return(0);
821: }
822:
823: // Method to reset the assembly channel
824:
825: void TAssemblyChannel::Reset() {
826:     value = BITC;
827:     valid = true;
828:     AsmChannelWMSelect = AsmChannelWMNone;
829:     AsmChannelZonesSelect = AsmChannelZonesNone;
830:     AsmChannelInvertSign = false;
831:     AsmChannelSignSelect = AsmChannelSignNone;
832:     AsmChannelNumericSelect = AsmChannelNumNone;
833: }
834:
835: // Common operation: Gate A Channel to Assembly Channel
836:
837: BCD TAssemblyChannel::GateAChannelToAssembly(BCD v) {
838:     AsmChannelWMSelect = AsmChannelWMA;
839:     AsmChannelZonesSelect = AsmChannelZonesA;
840:     AsmChannelInvertSign = false;
841:     AsmChannelSignSelect = AsmChannelSignNone;
842:     AsmChannelNumericSelect = AsmChannelNumA;
843:     value = v;      // Usually this will be an AChannel call return val
844:     valid = true;
845:     return(v);
846: }
847:
848: BCD TAssemblyChannel::GateBRegToAssembly(BCD v) {
849:     AsmChannelWMSelect = AsmChannelWMB;
850:     AsmChannelZonesSelect = AsmChannelZonesB;
851:     AsmChannelInvertSign = false;
```

```
852:         AsmChannelSignSelect = AsmChannelSignNone;
853:         AsmChannelNumericSelect = AsmChannelNumB;
854:         value = v;           // Usually this will be a B_Reg call return val
855:         valid = true;
856:         return(v);
857:     }
858:
859: }
860:
861: // Implementation of I/O Channel Class
862:
863: // Constructor.  Initializes state
864:
865: T1410Channel::T1410Channel(
866:     TLabel *LampInterlock,
867:     TLabel *LampRBCInterlock,
868:     TLabel *LampRead,
869:     TLabel *LampWrite,
870:     TLabel *LampOverlap,
871:     TLabel *LampNotOverlap,
872:     TLabel *LampNotReady,
873:     TLabel *LampBusy,
874:     TLabel *LampDataCheck,
875:     TLabel *LampCondition,
876:     TLabel *LampWLRecord,
877:     TLabel *LampNoTransfer ) {
878:
879:     ChStatus = 0;
880:     TapeDensity = DENSITY_200_556;
881:
882:     ChStatusDisplay[IOLAMPNOTREADY] = LampNotReady;
883:     ChStatusDisplay[IOLAMPBUSY] = LampBusy;
884:     ChStatusDisplay[IOLAMPDATACHECK] = LampDataCheck;
885:     ChStatusDisplay[IOLAMPCONDITION] = LampCondition;
886:     ChStatusDisplay[IOLAMPNOTTRANSFER] = LampNoTransfer;
887:     ChStatusDisplay[IOLAMPWLRECORD] = LampWLRecord;
888:
889:     // Generally, the channel latches are *not* reset by Program Reset
890:
891:     ChInterlock = new TDisplayLatch(LampInterlock, false);
892:     ChRBCInterlock = new TDisplayLatch(LampRBCInterlock, false);
893:     ChRead = new TDisplayLatch(LampRead, false);
894:     ChWrite = new TDisplayLatch(LampWrite, false);
895:     ChOverlap = new TDisplayLatch(LampOverlap, false);
896:     ChNotOverlap = new TDisplayLatch(LampNotOverlap, false);
897:
898:     // Generally, the channel registers are *not* reset by Program Reset
899:
900:     ChOp = new TRegister(false);
901:     ChUnitType = new TRegister(false);
902:     ChUnitNumber = new TRegister(false);
903:     ChR1 = new TRegister(false);
904:     ChR2 = new TRegister(false);
905: }
906:
907: // Channel is reset during ComputerReset
908:
909: void T1410Channel::OnComputerReset()
910: {
911:     ChStatus = 0;
912:
913:     // Note: The objects which are TDisplayLatch objects will reset themselves
914: }
915:
916: // Channel is not reset during Program Reset
917:
```

```
918: void T1410Channel::OnProgramReset()
919: {
920:     // Channel not affected by Program Reset
921: }
922:
923: // Display Routine.
924:
925: void T1410Channel::Display() {
926:
927:     int i;
928:
929:     ChStatusDisplay[IOLAMPNOTREADY] -> Enabled =
930:         ((ChStatus & IOCHNOTREADY) != 0);
931:     ChStatusDisplay[IOLAMPBUSY] -> Enabled =
932:         ((ChStatus & IOCHBUSY) != 0);
933:     ChStatusDisplay[IOLAMPDATACHECK] -> Enabled =
934:         ((ChStatus & IOCHDATACHECK) != 0);
935:     ChStatusDisplay[IOLAMPCCONDITION] -> Enabled =
936:         ((ChStatus & IOCHCONDITION) != 0);
937:     ChStatusDisplay[IOLAMPWLRECORD] -> Enabled =
938:         ((ChStatus & IOCHWLRECORD) != 0);
939:     ChStatusDisplay[IOLAMPNOTTRANSFER] -> Enabled =
940:         ((ChStatus & IOCHNOTTRANSFER) != 0);
941:
942:     for(i=0; i <= 5; ++i) {
943:         ChStatusDisplay[i] -> Repaint();
944:     }
945:
946:     // Although in most instances the following would be redundant,
947:     // because these objects are also on the CPU display list, we include
948:     // them here in case we want to display a channel separately.
949:
950:     ChInterlock -> Display();
951:     ChRBCInterlock -> Display();
952:     ChRead -> Display();
953:     ChWrite -> Display();
954:     ChOverlap -> Display();
955:     ChNotOverlap -> Display();
956: }
957:
958: // Channel Lamp Test
959:
960: void T1410Channel::LampTest(bool b)
961: {
962:     int i;
963:
964:     // Note, we don't have to do anything to the TDisplayLatch objects in
965:     // the channel for lamp test. They will take care of themselves on a
966:     // lamp test.
967:
968:     if(!b) {
969:         for(i=0; i <= 5; ++i) {
970:             ChStatusDisplay[i] -> Enabled = true;
971:             ChStatusDisplay[i] -> Repaint();
972:         }
973:     }
974:     else {
975:         Display();
976:     }
977: }
978:
979: □
980:
981: // CPU object constructor. Essentially this method "wires" the 1410.
982:
983: T1410CPU::T1410CPU()
```

```
984: {
985:     long i;
986:     TLabel *tmper,**tmpl;
987:
988:     CPU = this;
989:
990:     // Build the op code execute table
991:
992:     for(i=0; i < 64; ++i) {
993:         InstructionExecuteRoutine[i] = this -> InstructionExecuteInvalid;
994:     }
995:
996:     InstructionExecuteRoutine[OP_ADD] = InstructionArith;
997:     InstructionExecuteRoutine[OP_SUBTRACT] = InstructionArith;
998:     InstructionExecuteRoutine[OP_ZERO_ADD] = InstructionZeroArith;
999:     InstructionExecuteRoutine[OP_ZERO_SUB] = InstructionZeroArith;
1000:
1001:
1002:     // Clear out the lists
1003:
1004:     DisplayList = 0;
1005:     ResetList = 0;
1006:
1007:     // Set switches to initial states
1008:
1009:     Mode = MODE_RUN;
1010:     AddressEntry = ADDR_ENTRY_I;
1011:     StorageScan = SSCAN_OFF;
1012:     CycleControl = CYCLE_OFF;
1013:     CheckControl = CHECK_STOP;
1014:     DiskWrInhibit = false;
1015:     AsteriskInsert = true;
1016:     InhibitPrintOut = false;
1017:     BitSwitches = BCD(0);
1018:
1019:     // Build the various displayable components of the CPU
1020:
1021:     IRing = new TRingCounter(13);
1022:     assert(F1415L -> Light_I_OP != 0);
1023:     IRing -> lamps[0] = F1415L -> Light_I_OP;
1024:     IRing -> lamps[1] = F1415L -> Light_I_1;
1025:     IRing -> lamps[2] = F1415L -> Light_I_2;
1026:     IRing -> lamps[3] = F1415L -> Light_I_3;
1027:     IRing -> lamps[4] = F1415L -> Light_I_4;
1028:     IRing -> lamps[5] = F1415L -> Light_I_5;
1029:     IRing -> lamps[6] = F1415L -> Light_I_6;
1030:     IRing -> lamps[7] = F1415L -> Light_I_7;
1031:     IRing -> lamps[8] = F1415L -> Light_I_8;
1032:     IRing -> lamps[9] = F1415L -> Light_I_9;
1033:     IRing -> lamps[10] = F1415L -> Light_I_10;
1034:     IRing -> lamps[11] = F1415L -> Light_I_11;
1035:     IRing -> lamps[12] = F1415L -> Light_I_12;
1036:
1037:     ARing = new TRingCounter(6);
1038:     ARing -> lamps[0] = F1415L -> Light_A_1;
1039:     ARing -> lamps[1] = F1415L -> Light_A_2;
1040:     ARing -> lamps[2] = F1415L -> Light_A_3;
1041:     ARing -> lamps[3] = F1415L -> Light_A_4;
1042:     ARing -> lamps[4] = F1415L -> Light_A_5;
1043:     ARing -> lamps[5] = F1415L -> Light_A_6;
1044:
1045:     ClockRing = new TRingCounter(10);
1046:     ClockRing -> lamps[0] = F1415L -> Light_Clk_A;
1047:     ClockRing -> lamps[1] = F1415L -> Light_Clk_B;
1048:     ClockRing -> lamps[2] = F1415L -> Light_Clk_C;
1049:     ClockRing -> lamps[3] = F1415L -> Light_Clk_D;
```

```
1050:     ClockRing -> lamps[4] = F1415L -> Light_Clk_E;
1051:     ClockRing -> lamps[5] = F1415L -> Light_Clk_F;
1052:     ClockRing -> lamps[6] = F1415L -> Light_Clk_G;
1053:     ClockRing -> lamps[7] = F1415L -> Light_Clk_H;
1054:     ClockRing -> lamps[8] = F1415L -> Light_Clk_J;
1055:     ClockRing -> lamps[9] = F1415L -> Light_Clk_K;
1056:
1057:     ScanRing = new TRingCounter(4);
1058:     ScanRing -> lamps[0] = F1415L -> Light_Scan_N;
1059:     ScanRing -> lamps[1] = F1415L -> Light_Scan_1;
1060:     ScanRing -> lamps[2] = F1415L -> Light_Scan_2;
1061:     ScanRing -> lamps[3] = F1415L -> Light_Scan_3;
1062:
1063:     SubScanRing = new TRingCounter(5);
1064:     // NOTE: State 0 is "OFF" - no flip flops set
1065:     SubScanRing -> lamps[1] = F1415L -> Light_Sub_Scan_U;
1066:     SubScanRing -> lamps[2] = F1415L -> Light_Sub_Scan_B;
1067:     SubScanRing -> lamps[3] = F1415L -> Light_Sub_Scan_E;
1068:     SubScanRing -> lamps[4] = F1415L -> Light_Sub_Scan_MQ;
1069:
1070:     CycleRing = new TRingCounter(8);
1071:     CycleRing -> lamps[0] = F1415L -> Light_Cycle_A;
1072:     CycleRing -> lamps[1] = F1415L -> Light_Cycle_B;
1073:     CycleRing -> lamps[2] = F1415L -> Light_Cycle_C;
1074:     CycleRing -> lamps[3] = F1415L -> Light_Cycle_D;
1075:     CycleRing -> lamps[4] = F1415L -> Light_Cycle_E;
1076:     CycleRing -> lamps[5] = F1415L -> Light_Cycle_F;
1077:     CycleRing -> lamps[6] = F1415L -> Light_Cycle_I;
1078:     CycleRing -> lamps[7] = F1415L -> Light_Cycle_X;
1079:
1080:     // The Cycle Ring also displays on the CE panel - special case
1081:
1082:     CycleRing -> lampsCE = new TLabel*[8];
1083:     CycleRing -> lampsCE[0] = F1415L -> Light_CE_Cyc_A;
1084:     CycleRing -> lampsCE[1] = F1415L -> Light_CE_Cyc_B;
1085:     CycleRing -> lampsCE[2] = F1415L -> Light_CE_Cyc_C;
1086:     CycleRing -> lampsCE[3] = F1415L -> Light_CE_Cyc_D;
1087:     CycleRing -> lampsCE[4] = F1415L -> Light_CE_Cyc_E;
1088:     CycleRing -> lampsCE[5] = F1415L -> Light_CE_Cyc_F;
1089:     CycleRing -> lampsCE[6] = F1415L -> Light_CE_Cyc_I;
1090:     CycleRing -> lampsCE[7] = F1415L -> Light_CE_Cyc_X;
1091:
1092:
1093:     // Build the various latches. Most of these are not
1094:     // reset during Program Reset, but some are.
1095:
1096:     StopLatch = true;
1097:     StopKeyLatch = false;
1098:     DisplayModeLatch = false;
1099:     StorageWrapLatch = false;
1100:     ProcessRoutineLatch = false;
1101:     BranchLatch = true;
1102:     BranchTo1Latch = true;
1103:     LastInstructionReadout = false;
1104:     IRingControl = true;
1105:     IOChannelSelect = false;
1106:     IndexLatches = 0;
1107:
1108:     CarryIn = new TDisplayLatch(F1415L -> Light_Carry_In, false);
1109:     CarryOut = new TDisplayLatch(F1415L -> Light_Carry_Out, false);
1110:     AComplement = new TDisplayLatch(F1415L -> Light_A_Complement, false);
1111:     BComplement = new TDisplayLatch(F1415L -> Light_B_Complement, false);
1112:
1113:     CompareBGTA = new TDisplayLatch(F1415L -> Light_B_GT_A, false);
1114:     CompareBEQA = new TDisplayLatch(F1415L -> Light_B_EQ_A, false);
1115:     CompareBLTA = new TDisplayLatch(F1415L -> Light_B_LT_A, false);
```

```
1116:     Overflow = new TDisplayLatch(F1415L -> Light_Overflow, false);
1117:     DivideOverflow = new TDisplayLatch(F1415L -> Light_Divide_Overflow, false);
1118:     ZeroBalance = new TDisplayLatch(F1415L -> Light_Zero_Balance, false);
1119:
1120: // Build the various check latches. Program Reset does reset these
1121:
1122:     AChannelCheck = new TDisplayLatch(F1415L -> Light_Check_AChannel);
1123:     BChannelCheck = new TDisplayLatch(F1415L -> Light_Check_BChannel);
1124:     AssemblyChannelCheck = new TDisplayLatch(F1415L -> Light_Check_AssemblyChannel);
1125:     AddressChannelCheck = new TDisplayLatch(F1415L -> Light_Check_AddressChannel);
1126:     AddressExitCheck = new TDisplayLatch(F1415L -> Light_Check_AddressExit);
1127:     ARegisterSetCheck = new TDisplayLatch(F1415L -> Light_Check_ARegisterSet);
1128:     BRegisterSetCheck = new TDisplayLatch(F1415L -> Light_Check_BRegisterSet);
1129:     OpRegisterSetCheck = new TDisplayLatch(F1415L -> Light_Check_OpRegisterSet);
1130:     OpModifierSetCheck = new TDisplayLatch(F1415L -> Light_Check_OpModifierSet);
1131:     ACharacterSelectCheck = new TDisplayLatch(F1415L ->
1132:         Light_Check_ACharacterSelect);
1133:     BCharacterSelectCheck = new TDisplayLatch(F1415L ->
1134:         Light_Check_BCharacterSelect);
1135:     IOInterlockCheck = new TDisplayLatch(F1415L -> Light_Check_IOInterlock);
1136:     AddressCheck = new TDisplayLatch(F1415L -> Light_Check_AddressCheck);
1137:     RBCInterlockCheck = new TDisplayLatch(F1415L -> Light_Check_IOInterlock);
1138:     InstructionCheck = new TDisplayLatch(F1415L -> Light_Check_InstructionCheck);
1139:
1140: // Build the Data Registers
1141: A_Reg = new TRegister();
1142:
1143: tmper = F1415L -> Light_CE_A_ER;
1144: tmpl = new TLabel*[8];
1145: tmpl[0] = F1415L -> Light_CE_A_1;
1146: tmpl[1] = F1415L -> Light_CE_A_2;
1147: tmpl[2] = F1415L -> Light_CE_A_4;
1148: tmpl[3] = F1415L -> Light_CE_A_8;
1149: tmpl[4] = F1415L -> Light_CE_A_A;
1150: tmpl[5] = F1415L -> Light_CE_A_B;
1151: tmpl[6] = F1415L -> Light_CE_A_C;
1152: tmpl[7] = F1415L -> Light_CE_A_WM;
1153: A_Reg -> SetDisplay(tmper,tmpl);
1154:
1155: B_Reg = new TRegister();
1156:
1157: tmper = F1415L -> Light_CE_B_ER;
1158: tmpl = new TLabel*[8];
1159: tmpl[0] = F1415L -> Light_CE_B_1;
1160: tmpl[1] = F1415L -> Light_CE_B_2;
1161: tmpl[2] = F1415L -> Light_CE_B_4;
1162: tmpl[3] = F1415L -> Light_CE_B_8;
1163: tmpl[4] = F1415L -> Light_CE_B_A;
1164: tmpl[5] = F1415L -> Light_CE_B_B;
1165: tmpl[6] = F1415L -> Light_CE_B_C;
1166: tmpl[7] = F1415L -> Light_CE_B_WM;
1167: B_Reg -> SetDisplay(tmper,tmpl);
1168:
1169: Op_Reg = new TRegister();
1170:
1171: DEBUG("B_Reg is at %x",B_Reg)
1172: DEBUG("OP_Reg is at %x",Op_Reg)
1173:
1174: tmpl = new TLabel*[8];
1175: tmpl[0] = F1415L -> Light_CE_OP_1;
1176: tmpl[1] = F1415L -> Light_CE_OP_2;
1177: tmpl[2] = F1415L -> Light_CE_OP_4;
1178: tmpl[3] = F1415L -> Light_CE_OP_8;
1179: tmpl[4] = F1415L -> Light_CE_OP_A;
```

```
1180:     tmpl[5] = F1415L -> Light_CE_OP_B;
1181:     tmpl[6] = F1415L -> Light_CE_OP_C;
1182:     tmpl[7] = 0;
1183:     Op_Reg -> SetDisplay(0,tmpl);
1184:
1185:     Op_Mod_Reg = new TRegister();
1186:
1187:     tmpl = new TLabel*[8];
1188:     tmpl[0] = F1415L -> Light_CE_Mod_1;
1189:     tmpl[1] = F1415L -> Light_CE_Mod_2;
1190:     tmpl[2] = F1415L -> Light_CE_Mod_4;
1191:     tmpl[3] = F1415L -> Light_CE_Mod_8;
1192:     tmpl[4] = F1415L -> Light_CE_Mod_A;
1193:     tmpl[5] = F1415L -> Light_CE_Mod_B;
1194:     tmpl[6] = F1415L -> Light_CE_Mod_C;
1195:     tmpl[7] = 0;
1196:     Op_Mod_Reg -> SetDisplay(0,tmpl);
1197:
1198: // Build the Address Registers
1199:
1200: STAR = new TAddressRegister();
1201: A_AR = new TAddressRegister();
1202: B_AR = new TAddressRegister();
1203: C_AR = new TAddressRegister();
1204: D_AR = new TAddressRegister();
1205: E_AR = new TAddressRegister();
1206: F_AR = new TAddressRegister();
1207: I_AR = new TAddressRegister();
1208:
1209: // Build the channels
1210:
1211: Channel[CHANNEL1] = new T1410Channel(
1212:     F1415L -> Light_Ch1_Interlock,
1213:     F1415L -> Light_Ch1_RBCInterlock,
1214:     F1415L -> Light_Ch1_Read,
1215:     F1415L -> Light_Ch1_Write,
1216:     F1415L -> Light_Ch1_Overlap,
1217:     F1415L -> Light_Ch1_NoOverlap,
1218:     F1415L -> Light_Ch1_NotReady,
1219:     F1415L -> Light_Ch1_Busy,
1220:     F1415L -> Light_Ch1_DataCheck,
1221:     F1415L -> Light_Ch1_Condition,
1222:     F1415L -> Light_Ch1_WLRecord,
1223:     F1415L -> Light_Ch1_NoTransfer
1224):
1225:
1226: Channel[CHANNEL2] = new T1410Channel(
1227:     F1415L -> Light_Ch2_Interlock,
1228:     F1415L -> Light_Ch2_RBCInterlock,
1229:     F1415L -> Light_Ch2_Read,
1230:     F1415L -> Light_Ch2_Write,
1231:     F1415L -> Light_Ch2_Overlap,
1232:     F1415L -> Light_Ch2_NoOverlap,
1233:     F1415L -> Light_Ch2_NotReady,
1234:     F1415L -> Light_Ch2_Busy,
1235:     F1415L -> Light_Ch2_DataCheck,
1236:     F1415L -> Light_Ch2_Condition,
1237:     F1415L -> Light_Ch2_WLRecord,
1238:     F1415L -> Light_Ch2_NoTransfer
1239):
1240:
1241: // Build the A Channel and Assembly Channel
1242:
1243: AChannel = new TAChannel();
1244: AssemblyChannel = new TAssemblyChannel();
1245:
```

```
1246: // Some latches are set after power on...
1247:
1248: CompareBLTA -> Set();
1249: I_AR -> Set(1);
1250: AdderResult = 0;
1251:
1252: // Initialize core-load
1253:
1254: for(i=0; i < strlen(core_load_chars); ++i) {
1255:     core[i] = BCD::BCDConvert(core_load_chars[i]);
1256:     if(core_load_wm[i]) {
1257:         core[i].SetWM();
1258:     }
1259:     core[i].SetOddParity();
1260: }
1261:
1262: // Finally, set up the indicators
1263:
1264: OffNormal = new TDisplayIndicator(F1415L -> Light_Off_Normal,
1265:                                   &(CPU -> IndicatorOffNormal));
1266:
1267: }
1268:
1269:
1270: // CPU Object display - run thru the display list
1271:
1272: void T1410CPU::Display()
1273: {
1274:     TDisplayObject *l;
1275:
1276:     for(l = DisplayList; l != 0; l = l -> NextDisplay) {
1277:         l -> Display();
1278:     }
1279:
1280: }
1281:
1282: // Off Normal Indicator routine
1283:
1284: bool T1410CPU::IndicatorOffNormal()
1285: {
1286:     return(CPU -> InhibitPrintOut ||
1287:           !(CPU -> AsteriskInsert) ||
1288:           CPU -> CycleControl != CPU -> CYCLE_OFF ||
1289:           CPU -> CheckControl != CPU -> CHECK_STOP ||
1290:           (CPU -> Mode == CPU -> MODE_CE &&
1291:            CPU -> StorageScan != CPU -> SSCAN_OFF) ||
1292:            CPU -> AddressEntry != CPU -> ADDR_ENTRY_I );
1293: }
1294:
1295: // Cycle routine. Does typical kinds of CPU cycles
1296:
1297: void T1410CPU::Cycle()
1298: {
1299:     switch(CycleRing -> State()) {
1300:
1301:     case CYCLE_I:
1302:
1303:         *A_Reg = *B_Reg;
1304:         AssemblyChannel -> GateAChannelToAssembly(
1305:             AChannel -> Select(AChannel -> A_Channel_A));
1306:         break;
1307:
1308:     case CYCLE_X:
1309:         // This one is tricky, because it depends on opcode type.
1310:         // So this is handled in the indexing routines
1311:
```

```
1312:         break;
1313:
1314:     case CYCLE_A:
1315:         *A_Reg = *B_Reg;
1316:         AChannel -> Select(AChannel -> A_Channel_A);
1317:         A_AR -> Set(STARScan());
1318:         break;
1319:
1320:     case CYCLE_B:
1321:         B_AR -> Set(STARScan());
1322:         break;
1323:
1324:     default:
1325:         break;
1326:     }
1327: }
1328:
1329: // Storage routine to read out (access) core storage.
1330: // (A real core storage unit would also have to store it back)
1331: // Address is in the STAR (aka MAR)
1332:
1333: void T1410CPU::Readout()
1334: {
1335:     long i;
1336:
1337:     // Get the contents of STAR: The Memory Address Register
1338:
1339:     i = STAR -> Gate();           // Hee hee ;-)
1340:     StorageWrapLatch = false;
1341:
1342:     // This is really a debugging statement, but it's useful
1343:
1344:     if(i < 0) {
1345:         AddressCheck ->
1346:             SetStop("Address Check: STAR value not valid");
1347:         AddressExitCheck ->
1348:             SetStop("Address Exit Check: STAR value not valid");
1349:         B_Reg -> Set(0);           // Force a B Channel Check
1350:         return;
1351:     }
1352:
1353:     if(i > STORAGE) {
1354:         AddressCheck ->
1355:             SetStop("Address Check: STAR value > storage size");
1356:         B_Reg -> Set(0);
1357:         return;
1358:     }
1359:
1360:     // Check for a storage wrap condition. In most instances, it causes
1361:     // the CPU to stop with an Address Check, but there are exceptions.
1362:
1363:     if(i == STORAGE) {
1364:
1365:         // Valid wrap condition. Set address to 0, and set wrap condition.
1366:
1367:         i = 0;
1368:         StorageWrapLatch = true;
1369:     }
1370:
1371:     B_Reg -> Set(core[i]);    ✓
1372: }
1373:
1374: // Storage routine to store what is in the B Data Register
1375:
1376: void T1410CPU::Store()
1377: {
```

B Data Register

QCD bud

```

1378:     long i;
1379: 
1380:     // Get the contents of STAR: The Memory Address Register
1381: 
1382:     i = STAR -> Gate();
1383: 
1384:     // This is really a debugging statement, but it's useful
1385: 
1386:     if(i < 0) {
1387:         AddressCheck ->
1388:             SetStop("Address Check: STAR value not valid during store");
1389:         AddressExitCheck ->
1390:             SetStop("Address Exit Check: STAR value not valid during store");
1391:         return;
1392:     }
1393: 
1394:     if(i > STORAGE) {
1395:         AddressCheck ->
1396:             SetStop("Address Check: Star value > size of storage during store");
1397:         return;
1398:     }
1399: 
1400:     // Check for a storage wrap condition.
1401: 
1402:     if(i == STORAGE) {
1403: 
1404:         // Valid wrap condition. Set address to 0
1405: 
1406:         i = 0;
1407:         StorageWrapLatch = true;
1408:     }
1409: 
1410:     bcd
1411: 
1412:     if(!B_Reg -> Get()).CheckParity() {
1413:         BChannelCheck ->
1414:             SetStop("B Channel Check: Invalid parity");
1415:     }
1416: 
1417:     core[i] = B_Reg -> Get();
1418: }
1419: 
1420: // Storage routine to store data (Assembly Channel is implied here)
1421: 
1422: void T1410CPU::Store(BCD bcd)
1423: {
1424:     Set(bcd);
1425:     Remove
1426:     if(!B_Reg -> Get()).CheckParity() {
1427:         AssemblyChannelCheck ->
1428:             SetStop("Assembly Channel Check: B Channel invalid during store");
1429:     }
1430: 
1431:     Store();
1432: }
1433: 
1434: // Set Storage Scan Mode
1435: 
1436: void T1410CPU::SetScan(char i)
1437: {
1438:     ScanRing -> Set(i);
1439: }
1440: 
1441: // Apply storage scan value to STAR and return result.
1442: // Essentially this gates the ScanControl Ring to the Address Modification
1443: // circuitry

```

*more check*

```
1444:  
1445: long T1410CPU::STARScan()  
1446: {  
1447:     return(STAR -> Gate() + scan_mod[ScanRing -> State()]);  
1448: }  
1449:  
1450: // Apply +1 / 0 / -1 modification value to STAR and return result.  
1451: // This amounts to direct use of the address modification circuitry when  
1452: // not using SCAN CTRL (e.g. when doing Instruction readout)  
1453:  
1454: long T1410CPU::STARMod(int mod)  
1455: {  
1456:     return(STAR -> Gate() + mod);  
1457: }  
1458:  
1459: // Method to load core from a file  
1460:  
1461: void T1410CPU::LoadCore(char *filename)  
1462: {  
1463:     FILE *fd;  
1464:     long coresize,loc;  
1465:     char file_coresize[6];  
1466:     int file_core[STORAGE];  
1467:  
1468:     // First open the file.  
1469:  
1470:     if((fd = fopen(filename,"rb")) == NULL) {  
1471:         Application -> MessageBox("Unable to open file to load core.",  
1472:             "Load Core Error",MB_OK);  
1473:         return;  
1474:     }  
1475:  
1476:     // Next, read in the (5 digit) core size  
1477:  
1478:     file_coresize[sizeof(file_coresize)-1] = '\0';  
1479:     if(fread(&file_coresize,1,5,fd) != 5) {  
1480:         Application -> MessageBox("Error reading dump core size from file.",  
1481:             "Error Reading Dump",MB_OK);  
1482:         fclose(fd);  
1483:         return;  
1484:     }  
1485:  
1486:     // Convert from decimal to long, and validate.  
1487:  
1488:     coresize = atol(file_coresize);  
1489:     if(coresize < 0) {  
1490:         Application -> MessageBox("Dump contained negative value for core size.",  
1491:             "Negative Core Size in File",MB_OK);  
1492:         fclose(fd);  
1493:         return;  
1494:     }  
1495:  
1496:     if(coresize > STORAGE) {  
1497:         Application -> MessageBox("Dump contained core size greater than  
simulator's.",  
1498:             "Large Core Size in File",MB_OK);  
1499:         fclose(fd);  
1500:         return;  
1501:     }  
1502:  
1503:     // Read in the data from the dump file.  
1504:  
1505:     if(fread(&file_core,sizeof(int),coresize,fd) != coresize) {  
1506:         Application -> MessageBox("Error loading core data from file",  
1507:             "Error Loading Core",MB_OK);  
1508:         fclose(fd);
```

```
1509:     }
1510:
1511: // Finally, copy the data to core.
1512:
1513: for(loc=0; loc < coresize; ++loc) {
1514:     core[loc].Set(file_core[loc]);
1515: }
1516:
1517: fclose(fd);
1518: Application -> MessageBox("Core Loaded!","Core Loaded",MB_OK);
1519: }
1520:
1521: void T1410CPU::DumpCore(char *filename)
1522: {
1523:     FILE *fd;
1524:     long loc;
1525:     int file_core[STORAGE];
1526:
1527: // First, prepare a file.
1528:
1529: if((fd = fopen(filename,"wb")) == NULL) {
1530:     Application -> MessageBox("Unable to open file to dump core.",
1531:         "Dump Core Error",MB_OK);
1532: }
1533:
1534: // Then, copy core to an integer array
1535:
1536: for(loc=0; loc < STORAGE; ++loc){
1537:     file_core[loc] = core[loc].ToInt();
1538: }
1539:
1540: // Write out the core size to the file
1541:
1542: if(fprintf(fd,"%05d",STORAGE) != 5) {
1543:     Application -> MessageBox("Error writing out core size.",
1544:         "Core Size Write Error",MB_OK);
1545:     fclose(fd);
1546:     return;
1547: }
1548:
1549: // Then write out the integer array
1550:
1551: if(fwrite(&file_core,sizeof(int),STORAGE,fd) != STORAGE) {
1552:     Application -> MessageBox("Error writing out core file.",
1553:         "File Write Error",MB_OK);
1554:     fclose(fd);
1555: }
1556:
1557: Application -> MessageBox("Core Dumped!","Core Dumped",MB_OK);
1558: fclose(fd);
1559: }
1560:
1561: // The 1410 Adder. It works by translating the numeric part into
1562: // QuiBinary code. The real 1410 used logic gates to add, we use
1563: // a table.
1564:
1565: // Here is the code
1566:
1567: #define ADDER_BINARY_0 0
1568: #define ADDER_BINARY_1 1
1569: #define ADDER_QUINARY_0 0
1570: #define ADDER_QUINARY_1 1
1571: #define ADDER_QUINARY_2 2
1572: #define ADDER_QUINARY_3 3
1573: #define ADDER_QUINARY_4 4
1574:
```

```
1575: // This table translates the numeric part of a BCD character (8421) to
1576: // obtain the quinary part
1577:
1578: static int num_to_true_quinary[] = {
1579:     ADDER_QUINARY_0,                                // 0 (no bits)
1580:     ADDER_QUINARY_0,                                // 1
1581:     ADDER_QUINARY_2,                                // 2
1582:     ADDER_QUINARY_2,                                // 3 (2 + 1)
1583:     ADDER_QUINARY_4,                                // 4
1584:     ADDER_QUINARY_4,                                // 5 (4 + 1)
1585:     ADDER_QUINARY_6,                                // 6
1586:     ADDER_QUINARY_6,                                // 7 (6 + 1)
1587:     ADDER_QUINARY_8,                                // 8
1588:     ADDER_QUINARY_8,                                // 9 (8 + 1)
1589:     ADDER_QUINARY_0,                                // 0 (8 + 2)
1590:     ADDER_QUINARY_2,                                // 3 (2 + 1) (8 is ignored)
1591:     ADDER_QUINARY_4,                                // 4 (8 is ignored)
1592:     ADDER_QUINARY_4,                                // 5 (4+ 1) (8 is ignored)
1593:     ADDER_QUINARY_6,                                // 6 (8 is ingored)
1594:     ADDER_QUINARY_6                                // 7 (6 + 1) (8 is ignored)
1595: };
1596:
1597: // This table does the same thing, but generated the complement value,
1598: // used for subtraction and negative numbers. Note that these entries
1599: // are 8 - (the entry from the above table). But we are generated *indexes*
1600: // here, not actual values, so we couldn't just subtract!
1601:
1602: static int num_to_complement_quinary[] = {
1603:     ADDER_QUINARY_8, ADDER_QUINARY_8, ADDER_QUINARY_6, ADDER_QUINARY_6,
1604:     ADDER_QUINARY_4, ADDER_QUINARY_4, ADDER_QUINARY_2, ADDER_QUINARY_2,
1605:     ADDER_QUINARY_0, ADDER_QUINARY_0, ADDER_QUINARY_8, ADDER_QUINARY_6,
1606:     ADDER_QUINARY_4, ADDER_QUINARY_4, ADDER_QUINARY_2, ADDER_QUINARY_2
1607: };
1608:
1609: // Adder Quinary Matrix. Combines the two quinary values, returning
1610: // a quinary result, and a carry value. Indexed as [B][A], but the
1611: // table is symmetric (addition is complementary, even in quinary matrices!)
1612:
1613: struct adder_matrix {
1614:     int quinary;
1615:     bool carry;
1616: } adder_matrix[5][5] = {
1617:     { { ADDER_QUINARY_0, false }, { ADDER_QUINARY_2, false },      // 0-0, 0-2
1618:       { ADDER_QUINARY_4, false }, { ADDER_QUINARY_6, false },      // 0-4, 0-6
1619:       { ADDER_QUINARY_8, false } },
1620:
1621:     { { ADDER_QUINARY_2, false }, { ADDER_QUINARY_4, false },      // 2-0, 2-2
1622:       { ADDER_QUINARY_6, false }, { ADDER_QUINARY_8, false },      // 2-4, 2-6
1623:       { ADDER_QUINARY_0, true } },
1624:
1625:     { { ADDER_QUINARY_4, false }, { ADDER_QUINARY_6, false},      // 4-0, 4-2
1626:       { ADDER_QUINARY_8, false }, { ADDER_QUINARY_0, true },        // 4-4, 4-6
1627:       { ADDER_QUINARY_2, true } },
1628:
1629:     { { ADDER_QUINARY_6, false }, { ADDER_QUINARY_8, false },      // 6-0, 6-2
1630:       { ADDER_QUINARY_0, true }, { ADDER_QUINARY_2, true },        // 6-4, 6-6
1631:       { ADDER_QUINARY_4, true } },
1632:
1633:     { { ADDER_QUINARY_8, false }, { ADDER_QUINARY_0, true },        // 8-0, 8-2
1634:       { ADDER_QUINARY_2, true }, { ADDER_QUINARY_4, true },        // 8-4, 8-6
1635:       { ADDER_QUINARY_6, true } }
1636: };
1637:
1638: // Adder result table. Indexed via [quinary value][bshift value]
1639:
1640: int adder_result_table[5][4] = {
```

```
1641:     { 0, 1, 2, 3 },
1642:     { 2, 3, 4, 5 },
1643:     { 4, 5, 6, 7 },
1644:     { 6, 7, 8, 9 },
1645:     { 8, 9, 0, 1 }
1646: };
1647:
1648: // Adder.
1649:
1650: // Note: eventually we can make this more efficient by just generating
1651: // a 16x16 table for all of the possibilities during the CPU constructor!
1652:
1653: BCD T1410CPU::Adder(BCD A,int Complement_A,BCD B,int Complement_B)
1654: {
1655:     int bshift;
1656:     struct adder_matrix *qmatrix;
1657:     int bin;
1658:
1659:     int bcd_a,bcd_b;
1660:     int quinary_a,quinary_b;
1661:     int binary_a,binary_b;
1662:
1663:     bcd_a = A.ToInt();
1664:     bcd_b = B.ToInt();
1665:
1666:     // Based on the complement requests, translate BCD to QuiBinary.
1667:
1668:     if(Complement_A) {
1669:         quinary_a = num_to_complement_quinary[bcd_a & 0x0f];
1670:         binary_a = 1 - (bcd_a & 1);
1671:     }
1672:     else {
1673:         quinary_a = num_to_true_quinary[bcd_a & 0x0f];
1674:         binary_a = bcd_a & 1;
1675:     }
1676:
1677:     if(Complement_B) {
1678:         quinary_b = num_to_complement_quinary[bcd_b & 0x0f];
1679:         binary_b = 1 - (bcd_b & 1);
1680:     }
1681:     else {
1682:         quinary_b = num_to_true_quinary[bcd_b & 0x0f];
1683:         binary_b = bcd_b & 1;
1684:     }
1685:
1686:     // Handle the binary parts by adding them together along with the
1687:     // CarryIn latch. This results in a number from 0 to 3.
1688:
1689:     bshift = binary_a + binary_b + (CarryIn -> State());
1690:
1691:     // Add the Quinary parts together according to the matrix
1692:
1693:     qmatrix = &adder_matrix[quinary_a][quinary_b];
1694:
1695:     // Set the carry out state...
1696:
1697:     CarryOut -> Set(qmatrix -> carry ||
1698:                         (qmatrix -> quinary == ADDER_QUINARY_8 && bshift > 1));
1699:
1700:     // Calculate the binary result
1701:
1702:     bin = adder_result_table[qmatrix -> quinary][bshift];
1703:
1704:     // Finally, translate the binary result to BCD. Since we *know* that
1705:     // the bin value is range 0-9, we just cheat, and use the ascii to bcd
1706:     // table
```

```
1707:  
1708:     AdderResult = BCD::BCDConvert(bin + '0');  
1709:     return(AdderResult);  
1710: }
```

```

1: //-----
2: #include <vc1\vc1.h>
3: #pragma hdrstop
4:
5: #include "UI1410ARITH.h"
6: //-----
7:
8: #include <assert.h>
9:
10: #include "UI1410CPU.H"
11: #include "UI1410INST.H"
12: #include "UI1410DEBUG.H"
13:
14: // Arithmetic Instruction execution routines
15:
16: void T1410CPU::InstructionArith()
17: {
18:
19:     // Since arithmetic involves reading thru two fields, and since
20:     // we have to support STORAGE CYCLE, we need to remember which kind
21:     // we are expecting to do.
22:
23:     static int cycle_type;
24:     static int op_bin;
25:     int i;
26:     BCD adder_a, adder_b; latch adder_b adder_b hold_b           // What we feed to A side
27:     BCD adder_a, adder_b; latch adder_b adder_b hold_b
28:
29:     // If the first time in, set things up.
30:
31:     if>LastInstructionReadout) {
32:         SubScanRing -> Set(SUB_SCAN_U);           // Set the Units latch
33:         ScanRing -> Set(SCAN_1);                 // Set Scan to 1 (-1 modif.)
34:
35:         // Overflow is *not* reset! It stays on once set
36:
37:         CarryIn -> Reset();                      // Reset so it looks right in
38:         CarryOut -> Reset();                     // cycle testing
39:         AComplement -> Reset();                  // Here too - so that single
40:         BComplement -> Reset();                  // cycle looks nice.
41:
42:         cycle_type = CYCLE_A;                    // Set for initial A cycle
43:         op_bin = Op_Reg -> Get().ToInt() & 0x3f;
44:         LastInstructionReadout = false;
45:         ZeroBalance -> Set();                   // Assume zero balance to start
46:     }
47:
48:     if(cycle_type == CYCLE_A) {
49:         CycleRing -> Set(CYCLE_A);            // Set up for an A cycle
50:         *STAR = *A_AR;                        // Initiate cycle
51:         Readout();                          // Address storage from AAR
52:         Cycle();                            // Set AAR using proper mod
53:         cycle_type = CYCLE_B;                // Next time, do B Cycle
54:         return;                             // Next time, LIRO is false
55:         // But units latch is set!
56:     }
57:
58:     CycleRing -> Set(CYCLE_B);            // Normally, do a B cycle
59:     if(SubScanRing -> State() == SUB_SCAN_U) {
60:         *STAR = *D_AR;                      // Read B units via DAR
61:     }
62:     else {
63:         *STAR = *B_AR;                      // others use BAR
64:     }
65:     Readout();                          // Address storage frrom BAR
66:

```

AC 01C  
 STOR 01C  
 B REG 01C

```

67:     // Note that we don't do the address modification cycle yet, as we
68:     // have to store the result back where BAR points!
69:
70:     // The following looks complicated on the flow chart in the
71:     // IBM CE Instructional materials 1411 Processing Unit Instructions
72:     // on page 11. But, the end result is that if you have an EVEN number
73:     // of: -A, -B and SUBTRACT you do a TRUE ADD cycle, and ODD number and
74:     // you do a COMPLEMENT ADD cycle
75:     adder_b = B_Reg → Get();
76:     if(ScanRing -> State() == SCAN_1) {           // Is 1st scan ring on?
77:         if(SubScanRing -> State() == SUB_SCAN_U) { // Units latch?
78:             BComplement -> Reset();                // Set True Add B latch
79:             i =
80:                 (A_Reg -> Get().IsMinus()) +
81:                 (B_Reg -> Get().IsMinus()) +
82:                 (op_bin == OP_SUBTRACT);
83:             if(i & 1) {
84:                 AComplement -> Set();                  // Odd #: complement add
85:                 CarryIn -> Set();
86:             }
87:             else {
88:                 AComplement -> Reset();                // Even #: true add
89:                 CarryIn -> Reset();
90:             }
91:
92:             adder_a = A_Reg -> Get();
93:
94:         } // End, units
95:         else {
96:             if(SubScanRing -> State() == SUB_SCAN_E) { // Extension latch?
97:                 adder_a = BCD_0;                         // Yes, gate '0'
98:                                         // (if a compl then 9)
99:             }
100:            else {
101:                adder_a = A_Reg -> Get();              // No, use A data
102:            }
103:        }
104:    } // End, SCAN 1
105:
106:    else {
107:        assert(ScanRing -> State() == SCAN_3);      // Else must be 3
108:        if(SubScanRing -> State() == SUB_SCAN_U) { // Units?
109:            b_temp = B_Reg -> Get();                // Yes. Invert sign
110:            adder_b = b_temp;
111:            if(b_temp & ~BIT_ZONE) |
112:                (b_temp.IsMinus() ? (BITA | BITB) : BITB);
113:            b_temp.SetOddParity();
114:            AComplement -> Reset();                  // Reset compl add
115:        }
116:        adder_a = BCD_0;
117:    }
118:
119:    // Run the right stuff thru the adder, and then thru the Assembly Channel
120:
121:    Adder(addr_a, AComplement -> State(),
122:          B_Reg -> Get(), BComplement -> State());
123:    adder_b
124:    if(ZeroBalance -> State() &&
125:       (AdderResult & BIT_NUM) != (BCD_0 & ~BITC)) {
126:        ZeroBalance -> Reset();
127:    }
128:    hold_b = B_Reg → Get(); B_Reg → Set(addr_b);
129:    AssemblyChannel -> Select(
130:        AssemblyChannel -> AsmChannelWMB,
131:        AssemblyChannel -> AsmChannelZonesB,
132:        false,

```

*Handwritten notes:*

- Line 75:  $\checkmark$  adder\_b = B\_Reg → Get();
- Line 76:  $\checkmark$  if(ScanRing -> State() == SCAN\_1) {
- Line 77:  $\checkmark$  if(SubScanRing -> State() == SUB\_SCAN\_U) {
- Line 78:  $\checkmark$  BComplement -> Reset();
- Line 80:  $\checkmark$  i = (A\_Reg -> Get().IsMinus()) +
- Line 81:  $\checkmark$  (B\_Reg -> Get().IsMinus()) +
- Line 82:  $\checkmark$  (op\_bin == OP\_SUBTRACT);
- Line 83:  $\checkmark$  if(i & 1) {
- Line 84:  $\checkmark$  AComplement -> Set();
- Line 85:  $\checkmark$  CarryIn -> Set();
- Line 87:  $\checkmark$  else {
- Line 88:  $\checkmark$  AComplement -> Reset();
- Line 89:  $\checkmark$  CarryIn -> Reset();
- Line 91:  $\checkmark$  adder\_a = A\_Reg -> Get();
- Line 94:  $\checkmark$  } // End, units
- Line 95:  $\checkmark$  else {
- Line 96:  $\checkmark$  if(SubScanRing -> State() == SUB\_SCAN\_E) {
- Line 97:  $\checkmark$  adder\_a = BCD\_0;
- Line 98:  $\checkmark$  // (if a compl then 9)
- Line 99:  $\checkmark$  }
- Line 100:  $\checkmark$  else {
- Line 101:  $\checkmark$  adder\_a = A\_Reg -> Get();
- Line 102:  $\checkmark$  }
- Line 103:  $\checkmark$  }
- Line 104:  $\checkmark$  } // End, SCAN 1
- Line 105:  $\checkmark$
- Line 106:  $\checkmark$  else {
- Line 107:  $\checkmark$  assert(ScanRing -> State() == SCAN\_3);
- Line 108:  $\checkmark$  if(SubScanRing -> State() == SUB\_SCAN\_U) {
- Line 109:  $\checkmark$  b\_temp = B\_Reg -> Get();
- Line 110:  $\checkmark$  adder\_b = b\_temp;
- Line 111:  $\checkmark$  if(b\_temp & ~BIT\_ZONE) |
- Line 112:  $\checkmark$  (b\_temp.IsMinus() ? (BITA | BITB) : BITB);
- Line 113:  $\checkmark$  b\_temp.SetOddParity();
- Line 114:  $\checkmark$  AComplement -> Reset();
- Line 115:  $\checkmark$  }
- Line 116:  $\checkmark$  adder\_a = BCD\_0;
- Line 117:  $\checkmark$  }
- Line 118:  $\checkmark$
- Line 119:  $\checkmark$  // Run the right stuff thru the adder, and then thru the Assembly Channel
- Line 120:  $\checkmark$
- Line 121:  $\checkmark$  Adder(addr\_a, AComplement -> State(),
- Line 122:  $\checkmark$  B\_Reg -> Get(), BComplement -> State());
- Line 123:  $\checkmark$  adder\_b
- Line 124:  $\checkmark$  if(ZeroBalance -> State() &&
- Line 125:  $\checkmark$  (AdderResult & BIT\_NUM) != (BCD\_0 & ~BITC)) {
- Line 126:  $\checkmark$  ZeroBalance -> Reset();
- Line 127:  $\checkmark$  hold\_b = B\_Reg → Get(); B\_Reg → Set(addr\_b);
- Line 128:  $\checkmark$  AssemblyChannel -> Select(
- Line 129:  $\checkmark$  AssemblyChannel -> AsmChannelWMB,
- Line 130:  $\checkmark$  AssemblyChannel -> AsmChannelZonesB,
- Line 131:  $\checkmark$  false,
- Line 132:  $\checkmark$

*Handwritten annotations:*

- Line 75:  $\checkmark$  adder\_b = B\_Reg → Get();
- Line 76:  $\checkmark$  if(ScanRing -> State() == SCAN\_1) {
- Line 77:  $\checkmark$  if(SubScanRing -> State() == SUB\_SCAN\_U) {
- Line 78:  $\checkmark$  BComplement -> Reset();
- Line 80:  $\checkmark$  i = (A\_Reg -> Get().IsMinus()) +
- Line 81:  $\checkmark$  (B\_Reg -> Get().IsMinus()) +
- Line 82:  $\checkmark$  (op\_bin == OP\_SUBTRACT);
- Line 83:  $\checkmark$  if(i & 1) {
- Line 84:  $\checkmark$  AComplement -> Set();
- Line 85:  $\checkmark$  CarryIn -> Set();
- Line 87:  $\checkmark$  else {
- Line 88:  $\checkmark$  AComplement -> Reset();
- Line 89:  $\checkmark$  CarryIn -> Reset();
- Line 91:  $\checkmark$  adder\_a = A\_Reg -> Get();
- Line 94:  $\checkmark$  } // End, units
- Line 95:  $\checkmark$  else {
- Line 96:  $\checkmark$  if(SubScanRing -> State() == SUB\_SCAN\_E) {
- Line 97:  $\checkmark$  adder\_a = BCD\_0;
- Line 98:  $\checkmark$  // (if a compl then 9)
- Line 99:  $\checkmark$  }
- Line 100:  $\checkmark$  else {
- Line 101:  $\checkmark$  adder\_a = A\_Reg -> Get();
- Line 102:  $\checkmark$  }
- Line 103:  $\checkmark$  }
- Line 104:  $\checkmark$  } // End, SCAN 1
- Line 105:  $\checkmark$
- Line 106:  $\checkmark$  else {
- Line 107:  $\checkmark$  assert(ScanRing -> State() == SCAN\_3);
- Line 108:  $\checkmark$  if(SubScanRing -> State() == SUB\_SCAN\_U) {
- Line 109:  $\checkmark$  b\_temp = B\_Reg -> Get();
- Line 110:  $\checkmark$  adder\_b = b\_temp;
- Line 111:  $\checkmark$  if(b\_temp & ~BIT\_ZONE) |
- Line 112:  $\checkmark$  (b\_temp.IsMinus() ? (BITA | BITB) : BITB);
- Line 113:  $\checkmark$  b\_temp.SetOddParity();
- Line 114:  $\checkmark$  AComplement -> Reset();
- Line 115:  $\checkmark$  }
- Line 116:  $\checkmark$  adder\_a = BCD\_0;
- Line 117:  $\checkmark$  }
- Line 118:  $\checkmark$
- Line 119:  $\checkmark$  // Run the right stuff thru the adder, and then thru the Assembly Channel
- Line 120:  $\checkmark$
- Line 121:  $\checkmark$  Adder(addr\_a, AComplement -> State(),
- Line 122:  $\checkmark$  B\_Reg -> Get(), BComplement -> State());
- Line 123:  $\checkmark$  adder\_b
- Line 124:  $\checkmark$  if(ZeroBalance -> State() &&
- Line 125:  $\checkmark$  (AdderResult & BIT\_NUM) != (BCD\_0 & ~BITC)) {
- Line 126:  $\checkmark$  ZeroBalance -> Reset();
- Line 127:  $\checkmark$  hold\_b = B\_Reg → Get(); B\_Reg → Set(addr\_b);
- Line 128:  $\checkmark$  AssemblyChannel -> Select(
- Line 129:  $\checkmark$  AssemblyChannel -> AsmChannelWMB,
- Line 130:  $\checkmark$  AssemblyChannel -> AsmChannelZonesB,
- Line 131:  $\checkmark$  false,
- Line 132:  $\checkmark$

*Handwritten annotations (continued):*

- Line 127:  $\checkmark$  hold\_b = B\_Reg → Get(); B\_Reg → Set(addr\_b);
- Line 128:  $\checkmark$  AssemblyChannel -> Select(
- Line 129:  $\checkmark$  AssemblyChannel -> AsmChannelWMB,
- Line 130:  $\checkmark$  AssemblyChannel -> AsmChannelZonesB,
- Line 131:  $\checkmark$  false,
- Line 132:  $\checkmark$

```

133:     AssemblyChannel -> AsmChannelSignNone,
134:     AssemblyChannel -> AsmChannelNumAdder
135:   );
136:   ← B_Reg → Set(hold-b);
137:   if(B_Reg -> Get().TestWM()) {
138:     // B Channel WM?
139:     if(ScanRing -> State() == SCAN_3) {
140:       ✓ Store(AssemblyChannel -> Select());
141:       ✓ Cycle();
142:       ✓ IRingControl = true;
143:       return;
144:     }
145:     if(AComplement -> State()) { // Complement add?
146:       if(CarryOut -> State()) { // Carry too?
147:         ✓ Store(AssemblyChannel -> Select());
148:         ✓ Cycle();
149:         ✓ IRingControl = true;
150:         return;
151:       }
152:     }
153:   }
154:   // What I *think* the following does is to reverse the B sign
155:   // where D_AR points, because SUB_SCAN_U causes the CPU to use
156:   // the DAR for the storage cycle, which starts a recomplement
157:   // cycle wherein the entire B field is recomplemented. This
158:   // can only happen if we are doing a complement add,
159:   // with no carry out.
160:   // Set units latch
161:   SubScanRing -> Set(SUB_SCAN_U); // Set 3rd scan latch
162:   ScanRing -> Set(SCAN_3); // Set B compl add
163:   BComplement -> Set(); // Set carry
164:   CarryIn -> Set(); // Store the last char
165:   ✓ Store(AssemblyChannel -> Select()); // Finish the B cycle
166:   ✓ Cycle(); // Re-enter at B cycle
167:   cycle_type = CYCLE_B;
168:   return;
169: }
170: }
171: else { // True add
172:   if(CarryOut -> State()) { // Set overflow on carry
173:     Overflow -> Set();
174:   }
175:   ✓ Store(AssemblyChannel -> Select()); // Store the last char
176:   ✓ Cycle(); // Modify B addr -1
177:   ✓ IRingControl = true; // Done!
178:   return;
179: }
180: }
181: } // End B Ch WM
182: else { // No B Ch WM
183:   CarryIn -> Set(CarryOut -> State()); // Set carry
184:   ✓ Store(AssemblyChannel -> Select()); // Store result char
185:   ✓ Cycle(); // Modify B address -1
186:   if(ScanRing -> State() == SCAN_1) { // Still 1st scan?
187:     if(A_Reg -> Get().TestWM()) { // A Ch WM?
188:       if(A_Reg -> Get().TestWM()) { // Yes -- set extension
189:         SubScanRing -> Set(SUB_SCAN_E); // No more A cycles!
190:         cycle_type = CYCLE_B; // continue on
191:         return;
192:       }
193:       else { // No A CH WM
194:         SubScanRing -> Set(SUB_SCAN_B); // Set to body
195:         cycle_type = CYCLE_A; // A cycle next
196:         return;
197:       }
198:     }
199:   }
200: }

```

Original. OC.

```
199:     else {
200:         assert(ScanRing -> State() == SCAN_3);           // Must be!
201:         SubScanRing -> Set(SUB_SCAN_E);                 // Still in extension
202:         cycle_type = CYCLE_B;                           // Which means no A cyc
203:         return;
204:     }
205: }
206: }
207:
208: void T1410CPU::InstructionZeroArith()
209: {
210:
211:     // Variable to maintain state between cycles
212:
213:     static int cycle_type;
214:     static int op_bin;
215:
216:     BCD a_temp,a_hold;
217:
218:     // If this is the first time in, set things up
219:
220:     if(LastInstructionReadout) {
221:         SubScanRing -> Set(SUB_SCAN_U);                  // Units
222:         ScanRing -> Set(SCAN_1);                      // Scan 1 (-1 mod)
223:
224:         AComplement -> Reset();
225:         BComplement -> Reset();
226:         CarryIn -> Reset();
227:         CarryOut -> Reset();
228:         ZeroBalance -> Set();                          // Assume 0 at start
229:
230:         cycle_type = CYCLE_A;                           // Set for 1st A cycle
231:         op_bin = Op_Reg -> Get().ToInt() & 0x3f;
232:         LastInstructionReadout = false;
233:     }
234:
235:     // Initial A cycle
236:
237:     if(cycle_type == CYCLE_A) {
238:         CycleRing -> Set(CYCLE_A);                   // Take an A Cycle
239:         *STAR = *A_AR;
240:         Readout();                                     // RO A field char.
241:         Cycle();                                       // Set AAR with mod
242:
243:         // Units or body latch Regen happens by itself in simulator
244:
245:         cycle_type = CYCLE_B;                         // Set for B cycle
246:         return;
247:     }
248:
249:     assert(cycle_type == CYCLE_B);                   // Has to be B cycle
250:
251:     CycleRing -> Set(CYCLE_B);                   // Take B Cycle
252:     if(SubScanRing -> State() == SUB_SCAN_U) {      // Read B units via DAR
253:         *STAR = *D_AR;
254:     }
255:     else {
256:         *STAR = *B_AR;
257:     }
258:     Readout();                                     // Ro B Field Char
259:
260:     // If we are in the body/extension, we just insert 0's on the
261:     // A channel. Otherwise, we normalize the A field sign
262:
263:     a_temp = A_Reg -> Get();                      // Save to restore ...
264:     a_hold = a_temp;
```

```

265:
266:     if(SubScanRing -> State() == SUB_SCAN_U) {
267:         if(op_bin == OP_ZERO_ADD) {                                // ZA - normalize sign
268:             a_temp = (a_temp & ~BIT_ZONE) |
269:                     (a_temp.IsMinus() ? BITB : (BITA | BITB));
270:         }
271:     else {                                                 // ZS - invert sign
272:         a_temp = (a_temp & ~BIT_ZONE) |
273:                     (a_temp.IsMinus() ? (BITA | BITB) : BITB);
274:     }
275:     a_temp.SetOddParity();
276:
277:     // In the real machine, the sign fixing happens on the A Channel,
278:     // and not to the A Register itself. We'll cheat and use the A
279:     // register itself, so that the assembly channel can use the zones.
280:     // Then we will reset it later.
281:
282:     A_Reg -> Set(a_temp);
283: }
284:
285: if(SubScanRing -> State() == SUB_SCAN_E) {
286:     a_temp = BCD_0;
287: }
288:
289: // Run it thru the adder (kind of pointless in the simulator. Oh well.
290:
291: Adder(a_temp, false, BCD_0, false);
292:
293: if(ZeroBalance -> State() &&
294:     (AdderResult & BIT_NUM) != (BCD_0 & ~BITC)) {
295:     ZeroBalance -> Reset();
296: }
297:
298: AssemblyChannel -> Select(
299:     AssemblyChannel -> AsmChannelWMB,
300:     (SubScanRing -> State() == SUB_SCAN_U ?
301:         AssemblyChannel -> AsmChannelZonesA :
302:         AssemblyChannel -> AsmChannelZonesNone),
303:     false,
304:     AssemblyChannel -> AsmChannelSignNone,           // Maybe signlatch someday
305:     AssemblyChannel -> AsmChannelNumAdder
306: );
307:
308: ✓ Store(AssemblyChannel -> Select());                // Store the results
309: Cycle();                                              // Finish the cycle
310:
311: ✓ A_Reg -> Set(a_hold);                            // Reset a register
312:
313: ✓ if(B_Reg -> Get().TestWM()) {    OR, since it is
314:     IRingControl = true;          == org B in.
315:     return;                      {unchanged}
316: }
317:
318: // Again, 1st Scan Latch and True Add latch regen all by themselves
319:
320: CarryIn -> Set(CarryOut -> State());               // Do carry as needed
321:
322: if(a_hold.TestWM()) {                                // A Channel WM?
323:     SubScanRing -> Set(SUB_SCAN_E);                 // Yes -- extension
324:     cycle_type = CYCLE_B;                           // B Cycle Next
325:     return;
326: }
327: else {
328:     SubScanRing -> Set(SUB_SCAN_B);                 // No -- still body
329:     cycle_type = CYCLE_A;                           // A Cycle next
330:     return;

```

*(Handwritten notes)*

- ✓ Store(AssemblyChannel -> Select()); // Store the results
- ✓ Cycle(); // Finish the cycle
- ✓ A\_Reg -> Set(a\_hold); // Reset a register
- ✓ if(B\_Reg -> Get().TestWM()) { OR, since it is
  - = org B in.
  - {unchanged}
} // Done!

```

331:     }
332: }
333:
334: // Multiply instruction execution routine
335:
336: void T1410CPU::InstructionMultiply()
337: {
338:     static int cycle_type;
339:     static int cycle_subtype;
340:     static bool MultiplyDivideLastLatch;
341:
342:     BCD adder_a,b_temp;
343:
344:     if(LastInstructionReadout) {
345:         SubScanRing -> Set(SUB_SCAN_U);           // Set the Units latch
346:         ScanRing -> Set(SCAN_1);                // Set Scan to 1
347:         BComplement -> Reset();                  // True Add B
348:         AComplement -> Reset();                  // True Add A
349:         ZeroBalance -> Set();                   // Start as 0
350:         CarryIn -> Reset();                    // So single cycle
351:         CarryOut -> Reset();                   // looks good
352:         MultiplyDivideLastLatch = false;        // MDL latch reset
353:         SignLatch = false;                     // Start out as +
354:         cycle_type = CYCLE_A;                  // Start with A cycle
355:         cycle_subtype = 0;                      // No subtype
356:     } LastInstructionReadout = false;
357:
358:     if(cycle_type == CYCLE_A && cycle_subtype == 0) {
359:         CycleRing -> Set(CYCLE_A);            // Set for an A cycle
360:         if(SubScanRing -> State() == SUB_SCAN_U ||
361:             SubScanRing -> State() == SUB_SCAN_E) {
362:             *STAR = *C_AR;                      // A units, extension via CAR
363:         }
364:         else {
365:             assert(SubScanRing -> State() == SUB_SCAN_B); // A body via AAR
366:             *STAR = *A_AR;
367:         }
368:         Readout();
369:         Cycle();
370:         assert(ScanRing -> State() == SCAN_1); // Regen 1st Scan
371:         cycle_type = CYCLE_B;                  // Next cycle: B
372:         cycle_subtype = 0;
373:         return;
374:     } // End Cycle A
375:
376:     if(cycle_type == CYCLE_B && cycle_subtype == 0) {
377:
378:         CycleRing -> Set(CYCLE_B);            // Set for a B cycle
379:         if(SubScanRing -> State() == SUB_SCAN_U) {
380:             *STAR = *D_AR;                      // Read B units - DAR
381:         }
382:         else {
383:             *STAR = *B_AR;                      // Otherwise, use BAR
384:         }
385:         Readout();
386:
387:         if(SubScanRing -> State() == SUB_SCAN_U ||
388:             SubScanRing -> State() == SUB_SCAN_B ||
389:             SubScanRing -> State() == SUB_SCAN_E) {
390:
391:             Adder(BCD_0,AComplement->State(),      // Set up 0 in adder
392:                   BCD_0,BComplement->State());
393:             AssemblyChannel -> Select(          // No zones
394:                 AssemblyChannel -> AsmChannelWMB,
395:                 AssemblyChannel -> AsmChannelZonesNone,
396:                 false,
```

```

397:             AssemblyChannel -> AsmChannelSignNone,
398:             AssemblyChannel -> AsmChannelNumAdder);
399:             ✓ Store(AssemblyChannel -> Select()); // Store the 0
400:             // Modify B address by -1
401:             Cycle(); // Regen True add
402:             assert(!AComplement -> State()); // regen True add
403:             if(SubScanRing -> State() == SUB_SCAN_E) { // Extension
404:                 CarryIn -> Reset(); // No carry
405:                 SubScanRing -> Set(SUB_SCAN_MQ); // Set MQ latch
406:                 assert(ScanRing -> State() == SCAN_1); // Regen 1st scan
407:                 cycle_type = CYCLE_B; // B Cycle next
408:                 cycle_subtype = 0;
409:                 return;
410:             } // Extension
411:             assert(SubScanRing -> State() == SUB_SCAN_U || // Units or Body
412:                   SubScanRing -> State() == SUB_SCAN_B);
413:             if(A_Reg -> Get().TestWM()) { // A Channel WM?
414:                 SubScanRing -> Set(SUB_SCAN_E); // Set Extension latch
415:                 ScanRing -> Set(SCAN_N); // Set No Scan
416:                 cycle_type = CYCLE_C; // C Cycle next
417:                 cycle_subtype = 0;
418:                 return;
419:             }
420:             else {
421:                 SubScanRing -> Set(SUB_SCAN_B); // Set Body latch
422:                 assert(ScanRing -> State() == SCAN_1); // Regen 1st scan
423:                 cycle_type = CYCLE_A; // A Cycle next
424:                 cycle_subtype = 0;
425:                 return;
426:             } // End: Units or body
427:         } // End: Units, body or Extension
428:     } // End: Units, body or Extension
429:     else {
430:         assert(SubScanRing -> State() == SUB_SCAN_MQ); // MQ
431:         // As with add and subtract, the following is complicated on
432:         // the flow chart, but all it is really doing is checking to
433:         // see if the number of "-" values is even or odd
434:         if(((A_Reg -> Get().IsMinus()) + (B_Reg -> Get().IsMinus())) & 1) {
435:             CPU -> SignLatch = true; // Odd: Set - sign
436:         }
437:         else {
438:             CPU -> SignLatch = false;
439:         }
440:         DEBUG("Multiply, B Cycle, Sign is %d .",CPU -> SignLatch);
441:         adder_a = BCD_9; // 9 on adder A ch.
442:         b_temp = B_Reg -> Get(); // Analyze B ch char
443:         b_temp = b_temp & BIT_NUM; // Look at just numerics
444:         if(b_temp == (BCD_0 & BIT_NUM)) { // Store 0 in B Field
445:             ✓ Store(AssemblyChannel -> Select (
446:                 AssemblyChannel -> AsmChannelWMB,
447:                 AssemblyChannel -> AsmChannelZonesNone,
448:                 false,
449:                 AssemblyChannel -> AsmChannelSignNone,
450:                 AssemblyChannel -> AsmChannelNumZero));
451:             Cycle();
452:             AComplement -> Reset(); // Set True add latch
453:             CarryIn -> Reset(); // Set No Carry
454:             ✓ if(B_Reg -> Get().TestWM()) { // Set 2nd Scan
455:                 ScanRing -> Set(SCAN_2);
456:             }
457:         }
458:     }
459: 
```

```

463:             }
464:             else {
465:                 assert(ScanRing -> State() == SCAN_1); // Regen 1st scan
466:             }
467:         }
468:         else if(b_temp.ToInt() > 4) { // 5 - 9
469:             Store ✓ AssemblyChannel -> Select ( // Store B numerics
470:                 AssemblyChannel -> AsmChannelWMB,
471:                 AssemblyChannel -> AsmChannelZonesNone,
472:                 false,
473:                 AssemblyChannel -> AsmChannelSignNone,
474:                 AssemblyChannel -> AsmChannelNumB) ;
475:             Cycle();
476:             AComplement -> Set(); // Set Complement
477:             CarryIn -> Set(); // Set Carry
478:             ScanRing -> Set(SCAN_N); // Set No Scan
479:         }
480:
481:     else {
482:         Adder(adder_a,AComplement->State(),b_temp,BComplement->State());
483:         Store ✓ AssemblyChannel -> Select ( // Store adder output
484:                 AssemblyChannel -> AsmChannelWMB,
485:                 AssemblyChannel -> AsmChannelZonesNone,
486:                 false,
487:                 AssemblyChannel -> AsmChannelSignNone,
488:                 AssemblyChannel -> AsmChannelNumAdder) ;
489:             Cycle();
490:             AComplement -> Reset(); // Set True Latch
491:             CarryIn -> Reset(); // No Carry
492:             ScanRing -> Set(SCAN_N); // Set No Scan
493:         }
494:
495:         cycle_type = CYCLE_D;
496:         cycle_subtype = 0;
497:         return;
498:
499:     } // MQ
500:
501: } // End B Cycle, subtype 0
502:
503: if(cycle_type == CYCLE_C) {
504:     CycleRing -> Set(CYCLE_C); // C Cycle
505:     *STAR = *C_AR; // Read out units pos
506:     Readout(); // Read storage
507:     Cycle(); // C is not modified
508:     ScanRing -> Set(SCAN_1);
509:     cycle_type = CYCLE_B; // B Cycle next time
510:     cycle_subtype = 0;
511:     return;
512: } // End C Cycle
513:
514: if(cycle_type == CYCLE_D) {
515:     CycleRing -> Set(CYCLE_D); // D Cycle
516:     *STAR = *D_AR;
517:     Readout();
518:     if(ScanRing -> State() != SCAN_3) { // N, 1st or 2nd scan
519:         Store ✓ AssemblyChannel -> Select( // Store 0 & sign
520:                 AssemblyChannel -> AsmChannelWMB,
521:                 AssemblyChannel -> AsmChannelZonesNone,
522:                 false,
523:                 AssemblyChannel -> AsmChannelSignLatch,
524:                 AssemblyChannel -> AsmChannelNumZero) ;
525:     }
526:     // If 3rd Scan memory is regened (no store in emulator)
527:
528:     Cycle();

```

```

529:     CarryIn -> Set(AComplement -> State());           // Set carry state
530:
531:     if(ScanRing -> State() == SCAN_2) {                  // Done if 2nd Scan
532:         IRingControl = true;
533:         return;
534:     }
535:
536:     if(ScanRing -> State() == SCAN_N || MultiplyDivideLastLatch) {
537:         SubScanRing -> Set(SUB_SCAN_U);                   // Set Units latch
538:         ScanRing -> Set(SCAN_3);                         // Set 3rd scan
539:         BComplement -> Reset();                          // Set True Add B
540:         // AComplement regen'd
541:         cycle_type = CYCLE_A;                            // A Cycle next
542:         cycle_subtype = 1;                               // 2nd kind of A cycle
543:         return;
544:     }
545:
546:     // 1st or 3rd scan, MDL latch reset...
547:
548:     SubScanRing -> Set(SUB_SCAN_MQ);                   // Set MQ latch
549:     ScanRing -> Set(SCAN_3);                          // Set 3rd scan
550:     BComplement -> Reset();                          // True add B latch
551:     // Acomplement regen'd
552:     cycle_type = CYCLE_B;                            // B Cycle next
553:     cycle_subtype = 1;                               // 2nd kind of B cycle
554:     return;
555: }
556: // End D Cycle
557:
558: if(cycle_type == CYCLE_A && cycle_subtype == 1) {
559:     CycleRing -> Set(CYCLE_A);                      // A Cycle
560:     if(SubScanRing -> State() == SUB_SCAN_U || SubScanRing -> State() == SUB_SCAN_E) {
561:         *STAR = *C_AR;                                // Units or Extension?
562:         // Yes, use CAR for address
563:     }
564:     else {
565:         *STAR = *A_AR;
566:     }
567:     Readout();                                     // Read out A field
568:     Cycle();
569:     assert(ScanRing -> State() == SCAN_3);          // Regen 3rd scan
570:     BComplement -> Reset();                        // True add B latch
571:     // A Complement regen'd
572:     cycle_type = CYCLE_B;                          // B Cycle next
573:     cycle_subtype = 1;                            // 2nd kind of B cycle
574:     return;
575: }
576:
577: if(cycle_type == CYCLE_B && cycle_subtype == 1) {
578:
579:     CycleRing -> Set(CYCLE_B);                      // B Cycle
580:     if(SubScanRing -> State() == SUB_SCAN_U) {        // Read Units using DAR
581:         *STAR = *D_AR;
582:     }
583:     else {
584:         *STAR = *B_AR;
585:     }
586:     Readout();
587:
588:     if(SubScanRing -> State() == SUB_SCAN_MQ) {
589:         if(AComplement -> State()) {
590:             adder_a = BCD_0;                            // Insert 0 on A
591:             // With carry, will add
592:         }
593:     }
}

```

```

594:         adder_a = BCD_9;                                // Else insert a 9
595:     }
596:
597:     b_temp = B_Reg -> Get();                         // Get B data
598:     b_temp = b_temp & BIT_NUM;                        // Numeric bits
599:
600:     if(b_temp == (BCD_0 & BIT_NUM) && !(AComplement -> State()) ) {
601:       ✓ Store( AssemblyChannel -> Select (           // Store 0, no zones
602:               AssemblyChannel -> AsmChannelWMB,
603:               AssemblyChannel -> AsmChannelZonesNone,
604:               false,
605:               AssemblyChannel -> AsmChannelSignNone,
606:               AssemblyChannel -> AsmChannelNumZero) );
607:       Cycle();
608:       assert(! (AComplement -> State()));           // True Add (was already)
609:       CarryIn -> Reset();                            // Reset Carry
610:       ↳ if(B_Reg -> Get().TestWM()) {                // B WM?
611:           IRingControl = true;                         // Yes. All done.
612:           return;
613:       }
614:       assert(ScanRing -> State() == SCAN_3);        // Regen 3rd Scan latch
615:       assert(! (BComplement -> State()));           // Regen B True add
616:       cycle_type = CYCLE_D;                           // Next cycle D
617:       cycle_subtype = 0;
618:       return;
619:   } // End B is Zero, True Add
620:
621:   if(AComplement -> State() &&
622:       (b_temp == (BCD_0 & BIT_NUM) || b_temp.ToInt() < 5)) {
623:     ✓ Store ( AssemblyChannel -> Select (           // Store B (no zones)
624:               AssemblyChannel -> AsmChannelWMB,
625:               AssemblyChannel -> AsmChannelZonesNone,
626:               false,
627:               AssemblyChannel -> AsmChannelSignNone,
628:               AssemblyChannel -> AsmChannelNumB) );
629:     Cycle();
630:     AComplement -> Reset();                          // Set True Add latch
631:     CarryIn -> Reset();                            // Set Units latch
632:     SubScanRing -> Set(SUB_SCAN_U);                // Regen 3rd scan
633:     assert(ScanRing -> State() == SCAN_3);          // Set True Add B latch
634:     BComplement -> Reset();                         // Next Cycle: A
635:     cycle_type = CYCLE_A;                           // second kind of A
636:     cycle_subtype = 1;
637:     return;
638: } // End B is 0-4, Complement Add
639:
640: if(! (AComplement -> State()) && b_temp.ToInt() < 5) {
641:   Adder(addr_a,AComplement -> State(),
642:         B_Reg -> Get(),BComplement -> State());
643:   ↳ Store ( AssemblyChannel -> Select (           // Store adder, no zones
644:               AssemblyChannel -> AsmChannelWMB,
645:               AssemblyChannel -> AsmChannelZonesNone,
646:               false,
647:               AssemblyChannel -> AsmChannelSignNone,
648:               AssemblyChannel -> AsmChannelNumAdder) );
649:   Cycle();
650:   AComplement -> Reset();                          // Set True Add latch
651:   CarryIn -> Reset();                            // Set no carry
652:   SubScanRing -> Set(SUB_SCAN_U);                // Set units latch
653:   assert(ScanRing -> State() == SCAN_3);          // Regen 3rd scan
654:   BComplement -> Reset();                         // True add B
655:   cycle_type = CYCLE_A;                           // Next cycle A
656:   cycle_subtype = 1;                               // second kind of A
657:   return;
658: } // End B is 1-4, True Add
659:
```

```

660:         assert(b_temp != (BCD_0 & BIT_NUM) && b_temp.ToInt() >= 5);
661:
662:         if(b temp.ToInt() < 9 && AComplement -> State()) {
663:             Adder(addr_a, false, // DON'T USE AComplement
HERE!
664:             with // Because it was dealt
665:             // earlier
666:             ✓ B Reg -> Get(), BComplement -> State());
667:             ✓ Store(AssemblyChannel -> Select ( // Store adder, no zones
668:                 AssemblyChannel -> AsmChannelWMB,
669:                 AssemblyChannel -> AsmChannelZonesNone,
670:                 false,
671:                 AssemblyChannel -> AsmChannelSignNone,
672:                 AssemblyChannel -> AsmChannelNumAdder) );
673:             Cycle();
674:             AComplement -> Set(); // Set Complement latch
675:             CarryIn -> Set(); // Set carry latch
676:             SubScanRing -> Set(SUB_SCAN_U); // Set Units latch
677:             assert(ScanRing -> State() == SCAN_3); // Regen 3rd scan
678:             BComplement -> Reset(); // True add B
679:             cycle_type = CYCLE_A; // Next Cycle A
680:             cycle_subtype = 1; // second kind of A
681:             return;
682:         }
683:
684:         if(!(AComplement -> State())) { // True latch on?
685:             ✓ Store(AssemblyChannel -> Select ( // Store B, no zones
686:                 AssemblyChannel -> AsmChannelWMB,
687:                 AssemblyChannel -> AsmChannelZonesNone,
688:                 false,
689:                 AssemblyChannel -> AsmChannelSignNone,
690:                 AssemblyChannel -> AsmChannelNumB) );
691:             Cycle();
692:             AComplement -> Set(); // Set Complement latch
693:             CarryIn -> Set(); // Set carry latch
694:             SubScanRing -> Set(SUB_SCAN_U); // Set Units latch
695:             assert(ScanRing -> State() == SCAN_3); // Regen 3rd scan
696:             BComplement -> Reset(); // True add B
697:             cycle_type = CYCLE_A; // Next Cycle A
698:             cycle_subtype = 1; // second kind of A
699:             return;
700:         }
701:
702:         assert(b_temp.ToInt() == 9 && AComplement -> State()); // Store 0 or B, no zones
703:
704:         ✓ Store(AssemblyChannel -> Select ( // Store 0 or B, no zones
705:             AssemblyChannel -> AsmChannelWMB,
706:             AssemblyChannel -> AsmChannelZonesNone,
707:             false,
708:             AssemblyChannel -> AsmChannelSignNone,
709:             (ZeroBalance -> State() ?
710:                 AssemblyChannel -> AsmChannelNumZero :
711:                 AssemblyChannel -> AsmChannelNumB) );
712:             Cycle();
713:             ✓ if(B Reg -> Get().TestWM()) { // B WM?
714:                 AComplement -> Reset(); // Set True Add latch
715:                 MultiplyDivideLastLatch = true; // Set MDL latch
716:             }
717:             else {
718:                 AComplement -> Set(); // No. Set Complement
719:             }
720:             assert(ScanRing -> State() == SCAN_3); // Regen 3rd scan
721:             BComplement -> Reset(); // True Add B
722:             cycle_type = CYCLE_D; // D Cycle Next
723:             cycle_subtype = 0;

```



```
790:  
791:  
792: // Divide Instruction execute routine  
793:  
794: void T1410CPU::InstructionDivide()  
795: {  
796:  
797:     // Divide is using a new approach to state, so that when doing  
798:     // storage cycles, the storage print-out and indicators are the  
799:     // state of the machine at the end of the instruction. (The other  
800:     // instruction routines currently show scan and sub_scan for the  
801:     // *next* cycle, which confuses debugging.  
802:  
803:     static MultiplyDivideLastLatch;  
804:  
805:     static struct {  
806:         char cycle;  
807:         char subcycle;  
808:         char scan;  
809:         char subscan;  
810:     } next;  
811:  
812:     if(LastInstructionReadout) {  
813:         SubScanRing -> Set(SUB_SCAN_U);  
814:         next.subscan = SUB_SCAN_U;  
815:         ScanRing -> Set(SCAN_1);  
816:         next.scan = SCAN_1;  
817:         BComplement -> Reset();                                // True Add B  
818:         AComplement -> Set();                                 // Complement add A  
819:         CarryIn -> Set();                                    // NOT reset!  
820:         CarryOut -> Set();                                   // NOT reset!  
821:         MultiplyDivideLastLatch = false;  
822:         SignLatch = false;  
823:         next.cycle = CYCLE_A;                                // Entering A cycle  
824:         next.subcycle = 0;                                    // First kind  
825:     } LastInstructionReadout = false;  
826:  
827:     // Do common items for all cycle types  
828:  
829:     CycleRing -> Set(next.cycle);  
830:     ScanRing -> Set(next.scan);  
831:     SubScanRing -> Set(next.subscan);  
832:  
833:     // The "if" statements use the next.variables, for easy of coding..  
834:  
835:     if(next.cycle == CYCLE_A && next.subcycle == 0) {  
836:         assert(SubScanRing -> State() == SUB_SCAN_U);  
837:         *STAR = *C_AR;                                         // Use CAR to read units  
838:         Readout();                                            // Read into B register  
839:         Cycle();                                              // A Cycle copies to A Reg  
840:                                                 // And selects A Reg in  
841:                                                 // A Channel.  
842:         next.cycle = CYCLE_B;  
843:         next.subcycle = 0;  
844:         return;  
845:     }  
846:  
847:     if(next.cycle == CYCLE_B && next.subcycle == 0) {  
848:         if(SubScanRing -> State() == SUB_SCAN_U) {           // Units ?  
849:             *STAR = *D_AR;                                     // Use DAR to read units  
850:             Readout();  
851:             if(!(AComplement -> State()) &&  
852:                 ((B_Reg -> Get()) & BITB) != 0 ) {          // True add cycle?  
853:                 MultiplyDivideLastLatch = true;            // Yes. B Bit set?  
854:                 if((A_Reg -> Get().IsMinus()) !=      // Yes. Set MDL.  
855:                     ((A_Reg -> Get().IsMinus()) !=      // Different signs?
```

```

856:             (B_Reg->Get().IsMinus()) ) {           // Yes. Set minus latch
857:             CPU->SignLatch = true;
858:         }
859:         else {
860:             CPU->SignLatch = false;
861:         }
862:     }
863: }
864: else {                                // Not units...
865:     *STAR = *B_AR;                   // Use BAR for readout
866:     Readout();
867: } // End: Units
868:
869: if(SubScanRing->State() == SUB_SCAN_U ||      // Units or Body?
870:     SubScanRing->State() == SUB_SCAN_B) {
871:     Adder(A_Reg->Get(), AComplement->State(),      // Add A, B
872:            B_Reg->Get(), BComplement->State());
873:     OK Store(AssemblyChannel->Select (          // And store in *BAR
874:         AssemblyChannel->AsmChannelWMB,
875:         AssemblyChannel->AsmChannelZonesB,        // Don't disturb zones
876:         false,
877:         AssemblyChannel->AsmChannelSignNone,
878:         AssemblyChannel->AsmChannelNumAdder));
879:     Cycle();
880:     CarryIn->Set(CarryOut->State());           // Use adder to set carry
881:     if(A_Reg->Get().TestWM()) { .. b1c (6 cycles) // A Channel WM?
882:         next.subscan = SUB_SCAN_E;                 // Yes --> Extension
883:         return;
884:     }
885:     else {
886:         next.subscan = SUB_SCAN_B;                // No. Set Body
887:         next.cycle = CYCLE_A;                   // Next cycle A
888:         next.subcycle = 1;                      // Second kind
889:         return;
890:     }
891: } // End: Units, Body
892:
893: if(SubScanRing->State() == SUB_SCAN_E) {      // Extension?
894:
895:     // Add a 0 (or 9 if complement is on) to B, and store
896:     Adder(BCD_0, AComplement->State(),
897:            B_Reg->Get(), BComplement->State());
898:     OK Store(AssemblyChannel->Select (          // Should be no zones
899:         AssemblyChannel->AsmChannelWMB,
900:         AssemblyChannel->AsmChannelZonesB,        // AsmChannelZonesB
901:         false,
902:         AssemblyChannel->AsmChannelSignNone,
903:         AssemblyChannel->AsmChannelNumAdder));
904:     Cycle();
905:     CarryIn->Set(CarryOut->State());           // Set carry from adder
906:
907:     if(AComplement->State()) {                  // Complement cycle?
908:         if(CarryOut->State()) {                // Adder carry ?
909:             AComplement->Set();                  // Yes. Complement next
910:             next.subscan = SUB_SCAN_MQ;          // Set MQ latch
911:             next.cycle = CYCLE_B;              // B Cycle next
912:             next.subcycle = 0;                  // First kind.
913:             return;
914:         }
915:         else {                                // No adder carry.
916:             AComplement->Reset();            // True add next
917:             next.subscan = SUB_SCAN_U;          // Set Units latch
918:             next.scan = SCAN_3;                // Set 3rd scan
919:             next.cycle = CYCLE_A;              // A Cycle next
920:             next.subcycle = 1;                  // Second kind
921:             return;

```

```

922:         }
923:     } // End: Complement
924:
925:     if(MultiplyDivideLastLatch) { // Is MDL latch on?
926:         next.subscan = SUB_SCAN_MQ; // Set MQ Latch
927:         next.scan = SCAN_3; // Set 3rd scan
928:         AComplement -> Reset(); // True add next
929:         next.cycle = CYCLE_B; // B Cycle next
930:         next.subcycle = 0; // First kind
931:         return;
932:     }
933:     else { // No MDL latch
934:         next.scan = SCAN_2; // Set 2nd scan
935:         next.cycle = CYCLE_D; // D Cycle next
936:         next.subcycle = 0; // First kind
937:         return;
938:     }
939:
940: } // End: Extension
941:
942: assert(SubScanRing -> State() == SUB_SCAN_MQ); // Must be MQ
943:
944: if(AComplement -> State()) { // Complement ?
945:     assert(CarryIn -> State()); // Carry s/b on too
946:     Adder(BCD_0, false, B_Reg -> Get(), BComplement -> State()); // ++B
947:     Store(AssemblyChannel -> Select(
948:         AssemblyChannel -> AsmChannelWMB,
949:         AssemblyChannel -> AsmChannelZonesB, // Should be no zones
950:         false,
951:         AssemblyChannel -> AsmChannelSignNone,
952:         AssemblyChannel -> AsmChannelNumAdder));
953:     Cycle();
954:     if(CarryOut -> State()) { // Adder carry?
955:         DivideOverflow -> Set(); // Yes. Set overflow!
956:         IRingControl = true; // And be done.
957:         return;
958:     }
959:     CarryIn -> Set(); // No. But set carry now
960:     next.subscan = SUB_SCAN_U; // Set units latch
961:     next.scan = SCAN_3; // Set 3rd scan
962:     AComplement -> Set(); // Complement Add next
963:     next.cycle = CYCLE_A; // A cycle next
964:     next.subcycle = 1; // Second kind.
965:     return;
966: } // End: MQ, Complement
967:
968: assert(SubScanRing -> State() == SUB_SCAN_MQ && MultiplyDivideLastLatch);
969:
970: assert(!(AComplement -> State())); // Regen True Add
971: Adder(BCD_9, false, B_Reg -> Get(), BComplement -> State()); // --B
972: Store(AssemblyChannel -> Select( // Store sign in result
973:     AssemblyChannel -> AsmChannelWMB,
974:     AssemblyChannel -> AsmChannelZonesNone,
975:     false,
976:     AssemblyChannel -> AsmChannelSignLatch,
977:     AssemblyChannel -> AsmChannelNumAdder));
978: Cycle();
979: IRingControl = true; // And be done.
980: return;
981:
982: } // End: B Cycle, subtype 0
983:
984: if(next.cycle == CYCLE_D) { // D reg to RO B field
985:     *STAR = *D_AR;
986:     Readout();
987:     Cycle();

```

```
988:         next.subscan = SUB_SCAN_U;           // Set units latch
989:         next.scan = SCAN_3;                 // Set 3rd scan
990:         AComplement -> Set();            // Complement add
991:         CarryIn -> Set();              // Carry
992:         next.cycle = CYCLE_A;            // A Cycle next
993:         next.subcycle = 1;               // Second kind
994:         return;
995:     }
996:
997:     if(next.cycle == CYCLE_A && next.subcycle == 1) { // A cycle
998:         if(SubScanRing -> State() == SUB_SCAN_U) {
999:             *STAR = *C_AR;                  // Units RO via CAR
1000:        }
1001:        else {
1002:            assert(SubScanRing -> State() == SUB_SCAN_B); // Else s/b body
1003:            *STAR = *A_AR;
1004:        }
1005:        Readout();
1006:        Cycle();
1007:        next.cycle = CYCLE_B;            // B Cycle next
1008:        next.subcycle = 0;
1009:        return;
1010:    }
1011:
1012:    assert(false);                      // OOPS
1013: }
1014:
```

```
1: //  
2: // This Unit is the main part of the 1410 emulator  
3: //  
4:  
5: //-----  
6: #include <vc1\vc1.h>  
7: #pragma hdrstop  
8:  
9: #include "UI1410CPU.h"  
10: #include "UI1415IO.h"  
11: //-----  
12:  
13: // We have to predefine CPU, because the CPU object's children need a  
14: // pointer to the CPU to set up the various lists.  
15:  
16: T1410CPU *CPU = 0;  
17:  
18: // Construct and initialize the CPU.  
19:  
20: void Init1410()  
21: {  
22:     DEBUG("Creating CPU Object",0);  
23:  
24:     new T1410CPU;  
25:  
26:     // The following sets are for testing the Computer Reset button...  
27:  
28:     CPU -> IRing -> Set(I_RING_2);  
29:     CPU -> SubScanRing -> Set(SUB_SCAN_MQ);  
30:     CPU -> CycleRing -> Set(CYCLE_F);  
31:     CPU -> CarryIn -> Set();  
32:     CPU -> BComplement -> Set();  
33:     CPU -> CompareBLTA -> Reset();  
34:     CPU -> CompareBEQA -> Set();  
35:     CPU -> InstructionCheck -> Set();  
36:  
37:  
38:     // End of test sets  
39:  
40:     FI1415IO -> SetState(CONSOLE_IDLE);  
41:     CPU -> Display();  
42:  
43:     DEBUG("STAR is initially %s",CPU -> STAR -> IsValid() ? "Valid" : "Invalid");  
44:     DEBUG("Setting STAR from an int",0);  
45:     CPU -> STAR -> Set(12345);  
46:     DEBUG("STAR is now %s",CPU -> STAR -> IsValid() ? "Valid" : "Invalid");  
47:     DEBUG("STAR (int) value is now %d",CPU -> STAR -> Gate());  
48:     DEBUG("Resetting STAR",0);  
49:     CPU -> STAR -> Reset();  
50:     DEBUG("STAR is now %s",CPU -> STAR -> IsValid() ? "Valid" : "Invalid");  
51:     CPU -> STAR -> Set(1,1);  
52:     CPU -> STAR -> Set(2,2);  
53:     CPU -> STAR -> Set(3,3);  
54:     CPU -> STAR -> Set(4,4);  
55:     CPU -> STAR -> Set(5,5);  
56:     DEBUG("STAR is now %s",CPU -> STAR -> IsValid() ? "Valid" : "Invalid");  
57:     DEBUG("STAR (int) value is now %d",CPU -> STAR->Gate());  
58:  
59:     DEBUG("Waiting for User Interface",0)  
60:  
61:  
62: }  
63:
```

Register assignment test

```
1: #ifndef UBCDH
2: #define UBCDH
3:
4: //
5: //  Definition of character representation
6: //
7:
8: #define BITWM 0x80
9: #define BIT1 1
10: #define BIT2 2
11: #define BIT4 4
12: #define BIT8 8
13: #define BITA 0x10
14: #define BITB 0x20
15: #define BITC 0x40
16:
17: #define BIT_NUM 0x0f
18: #define BIT_ZONE 0x30
19:
20: extern char bcd_ascii[];
21: extern int ascii_bcd[];
22: extern int bcd_to_two_of_five_table[];
23: extern int two_of_five_to_bin_table[];
24: extern int bin_to_two_of_five_table[];
25: extern int parity_table[];
26: extern int odd_parity_table[];
27:
28: enum bcd_char_type { BCD_NN = 1, BCD_SC = 2, BCD_AN = 3 };
29:
30: extern bcd_char_type bcd_char_type_table[];
31:
32: //
33: //  These are defined so that keyboard input may recognize them
34: //
35:
36: #define ASCII_RECORD_MARK 0174
37: #define ASCII_GROUP_MARK 0316
38: #define ASCII_SEGMENT_MARK 0327
39: #define ASCII_RADICAL 0373
40: #define ASCII_ALT_BLANK 'b'
41: #define ASCII_WORD_SEPARATOR '^'
42: #define ASCII_DELTA 0177
43:
44: class BCD {
45:
46: private:
47:     int c; // WM C B A 8 4 2 1
48:
49: public:
50:
51:     BCD() { c = 0; } // Default constructor
52:
53:     BCD(int i) { c = i; }
54:
55:     BCD Set(int i) { c = i; }
56:
57:     static inline int BCDConvert(int ch) {
58:         if(ascii_bcd[ch] < 0) {
59:             return ascii_bcd[ASCII_ALT_BLANK];
60:         }
61:         else {
62:             return ascii_bcd[ch];
63:         }
64:     }
65:
66:     static inline int BCDCheck(int ch) {
```

```
67:         return(ascii_bcd[ch]);
68:     }
69:
70: //  Ascii translation ignores wordmark and parity bits.
71:
72: inline char ToAscii() {
73:     return bcd_ascii[c & 077];
74: }
75:
76: inline intToInt() {
77:     return c;
78: }
79:
80: inline bool IsMinus() {
81:     return((c & BIT_ZONE) == BITB);
82: }
83:
84: inline bool TestWM() {
85:     return((c & BITWM) != 0);
86: }
87:
88: inline bool TestCheck() {
89:     return((c & BITC) != 0);
90: }
91:
92: inline void SetWM() {
93:     c |= BITWM;
94: }
95:
96: inline void ClearWM() {
97:     c &= ~BITWM;
98: }
99:
100: inline void SetCheck() {
101:     c |= BITC;
102: }
103:
104: inline void ComplementCheck() {
105:     c ^= BITC;
106: }
107:
108: inline void ClearCheck() {
109:     c &= ~BITC;
110: }
111:
112: inline bool CheckParity() {
113:     return(odd_parity_table[c]);
114: }
115:
116: inline bool SetOddParity() {
117:     if(!odd_parity_table[c]) {
118:         c ^= BITC;
119:     }
120: }
121:
122: // Routine to test the parity of a BCD character
123: // Returns 0 for Even Parity, 1 for Odd Parity
124: // The test includes the WM bit, but does NOT itself
125: // include the check bit.
126:
127: inline int GetParity() { return(parity_table[c]); }
128:
129: // Overloaded standard bit operator functions
130:
131: BCD operator&(BCD bcd) {
132:     return(BCD(c & bcd.c));
```

```
133:     }
134:
135:     BCD operator&(int i) {
136:         return(BCD(c & i));
137:     }
138:
139:     BCD operator|(BCD bcd) {
140:         return(BCD(c | bcd.c));
141:     }
142:
143:     BCD operator|(int i) {
144:         return(BCD(c | i));
145:     }
146:
147:     BCD operator<<(int shift) {
148:         return(BCD(c << shift));
149:     }
150:
151:     BCD operator>>(int shift) {
152:         return(BCD(c >> shift));
153:     }
154:
155:     bool operator==(BCD bcd) {
156:         return(c == bcd.c);
157:     }
158:
159:     bool operator!=(BCD bcd) {
160:         return(c != bcd.c);
161:     }
162:
163:     bool operator==(int i) {
164:         return(c == i);
165:     }
166:
167:     bool operator!=(int i) {
168:         return(c != i);
169:     }
170:
171:     enum bcd_char_type GetType() {
172:         return bcd_char_type_table[c];
173:     }
174:
175: };
176:
177: // Class for the 2 of 5 code used in address registers. The "0 bit"
178: // (really a parity bit) is encoded as 16.
179:
180: #define TWOOF5_1      1
181: #define TWOOF5_2      2
182: #define TWOOF5_4      4
183: #define TWOOF5_8      8
184: #define TWOOF5_0     16
185:
186: class TWOOF5 {
187:
188: private:
189:     int b;
190:
191: public:
192:
193:     TWOOF5() { b = 0; }
194:     TWOOF5(BCD bcd) { b = bcd_to_two_of_five_table[bcd.ToInt() & 0x7f]; }
195:     TWOOF5(int i) { b = bin_to_two_of_five_table[i]; }
196:     inlineToInt() { return two_of_five_to_bin_table[b]; }
197:     inlineToBCD() {
198:         return(ascii_bcd['0' + ToInt()]);
199:
```

```
199:      };  
200: };  
201:  
202: #endif  
203:
```

```
1: //  
2: // This Unit provides the implementation of bcd characters  
3: //  
4:  
5: #include "ubcd.h"  
6:  
7: /*  
8:     The following tables were copied from Joseph  
9:     Newcomer's 1401 emulation, in order to provide  
10:    consistent card, tape and print facilities.  
11:  
12:    Jay Jaeger, 12/96  
13: */  
14:  
15: /* The following table is given in the order of the 1401 BCD codes, and  
16:     contains the equivalent ASCII codes for printout.  
17: */  
18: char bcd_ascii[64] = {  
19:     ' ', /* 0           - space */  
20:     '1', /* 1           1   - 1 */  
21:     '2', /* 2           2   - 2 */  
22:     '3', /* 3           21  - 3 */  
23:     '4', /* 4           4   - 4 */  
24:     '5', /* 5           41  - 5 */  
25:     '6', /* 6           42  - 6 */  
26:     '7', /* 7           421 - 7 */  
27:     '8', /* 8           8   - 8 */  
28:     '9', /* 9           8 1 - 9 */  
29:     '0', /* 10          8 2 - 0 */  
30:     '=', /* 11          8 21 - number sign (#) or equal */  
31:     '\'', /* 12          84  - at sign @ or quote */  
32:     ':', /* 13          84 1 - colon */  
33:     '>', /* 14          842 - greater than */  
34:     'û', /* 15          8421 - radical */  
35:     'b', /* 16          A   - substitute blank */  
36:     '/', /* 17          A 1 - slash */  
37:     'S', /* 18          A 2 - S */  
38:     'T', /* 19          A 21 - T */  
39:     'U', /* 20          A 4 - U */  
40:     'V', /* 21          A 4 1 - V */  
41:     'W', /* 22          A 42 - W */  
42:     'X', /* 23          A 421 - X */  
43:     'Y', /* 24          A8  - Y */  
44:     'Z', /* 25          A8 1 - Z */  
45:     '\174', /* 26          A8 2 - record mark */  
46:     ',', /* 27          A8 21 - comma */  
47:     '(', /* 28          A84 - percent % or paren */  
48:     '^', /* 29          A84 1 - word separator */  
49:     '\\\\', /* 30          A842 - left oblique */  
50:     'x', /* 31          A8421 - segment mark */  
51:     '-', /* 32          B   - hyphen */  
52:     'J', /* 33          B 1 - J */  
53:     'K', /* 34          B 2 - K */  
54:     'L', /* 35          B 21 - L */  
55:     'M', /* 36          B 4 - M */  
56:     'N', /* 37          B 4 1 - N */  
57:     'O', /* 38          B 42 - O */  
58:     'P', /* 39          B 421 - P */  
59:     'Q', /* 40          B 8 - Q */  
60:     'R', /* 41          B 8 1 - R */  
61:     '!', /* 42          B 8 2 - exclamation (-0) */  
62:     '$', /* 43          B 8 21 - dollar sign */  
63:     '*', /* 44          B 84 - asterisk */  
64:     ']', /* 45          B 84 1 - right bracket */  
65:     ';', /* 46          B 842 - semicolon */  
66:     '\177', /* 47          B 8421 - delta */
```

```

67:         '+' ,    /* 48 BA      - ampersand or plus */
68:         'A' ,    /* 49 BA 1   - A */
69:         'B' ,    /* 50 BA 2   - B */
70:         'C' ,    /* 51 BA 21  - C */
71:         'D' ,    /* 52 BA 4   - D */
72:         'E' ,    /* 53 BA 4 1 - E */
73:         'F' ,    /* 54 BA 42  - F */
74:         'G' ,    /* 55 BA 421 - G */
75:         'H' ,    /* 56 BA8   - H */
76:         'I' ,    /* 57 BA8 1  - I */
77:         '?' ,    /* 58 BA8 2  - question mark */
78:         '.' ,    /* 59 BA8 21 - period */
79:         ')' ,    /* 60 BA84  - lozenge or paren */
80:         '[' ,    /* 61 BA84 1 - left bracket */
81:         '<' ,    /* 62 BA842 - less than */
82:         ']' ,    /* 63 BA8421 - group mark */

83: };
84:
85:
86: ****
87: The following table is used to convert ASCII characters to BCD.
88:
89: Note that it currently is not complete.
90:
91: The following substitutions or alternate mappings are made:
92:
93:     ASCII code   BCD      Notes
94:     -----   ---      -----
95:     "        N/A      illegal
96:     %        (        'H' character set representation
97:     &       +        'H' character set representation
98:     @        '        'H' character set representation
99:     #        =        'H' character set representation
100:    ``       N/A      illegal
101:    ^        ^        substitute graphic for word-separator
102:    `       N/A      illegal
103:    a,c-z   A,C-Z    case folded
104:    b        b        substitute blank
105:    {        N/A      illegal
106:    }        N/A      illegal
107:    ~        N/A      illegal
108:    |        Ø        substitute for record mark
109:
110: ****
111:
112: int ascii_bcd[256] = {
113:     -1,-1,-1,-1,-1,-1,-1,-1,    /* 00 - 07 illegal */
114:     -1,-1,-1,-1,-1,-1,-1,-1,    /* 010 - 017 illegal */
115:     -1,-1,-1,-1,-1,-1,-1,-1,    /* 020 - 027 illegal */
116:     -1,-1,-1,-1,-1,-1,-1,-1,    /* 030 - 037 illegal */
117:
118:     0,          /* 040 space */
119:     42,         /* 041 ! */
120:     -1,         /* 042 " illegal */
121:     11,         /* 043 # */
122:     43,         /* 044 $ */
123:     28,         /* 045 % also ( */
124:     48,         /* 046 & also + */
125:     12,         /* 047 ' also @ */
126:
127:     28,         /* 050 ( also % */
128:     60,         /* 051 ) also lozenge */
129:     44,         /* 052 * */
130:     48,         /* 053 + also & */
131:     27,         /* 055 , */
132:     32,         /* 055 - */

```

```
133:    59,          /* 056 . */
134:    17,          /* 057 / */
135:
136:    10,          /* 060 0 */
137:    1,           /* 061 1 */
138:    2,           /* 062 2 */
139:    3,           /* 063 3 */
140:    4,           /* 064 4 */
141:    5,           /* 065 5 */
142:    6,           /* 066 6 */
143:    7,           /* 067 7 */
144:
145:    8,           /* 070 8 */
146:    9,           /* 071 9 */
147:   13,           /* 072 : */
148:   46,           /* 073 ; */
149:   62,           /* 074 < */
150:   11,           /* 075 = also # */
151:   14,           /* 076 > */
152:   58,           /* 077 ? */
153:
154:   12,           /* 0100 @ */
155:   49,           /* 0101 A */
156:   50,           /* 0102 B */
157:   51,           /* 0103 C */
158:   52,           /* 0104 D */
159:   53,           /* 0105 E */
160:   54,           /* 0106 F */
161:   55,           /* 0107 G */
162:
163:   56,           /* 0110 H */
164:   57,           /* 0111 I */
165:   33,           /* 0112 J */
166:   34,           /* 0113 K */
167:   35,           /* 0114 L */
168:   36,           /* 0115 M */
169:   37,           /* 0116 N */
170:   38,           /* 0117 O */
171:
172:   39,           /* 0120 P */
173:   40,           /* 0121 Q */
174:   41,           /* 0122 R */
175:   18,           /* 0123 S */
176:   19,           /* 0124 T */
177:   20,           /* 0125 U */
178:   21,           /* 0126 V */
179:   22,           /* 0127 W */
180:
181:   23,           /* 0130 X */
182:   24,           /* 0131 Y */
183:   25,           /* 0132 Z */
184:   61,           /* 0133 [ */
185:   30,           /* 0134 \ */
186:   45,           /* 0135 ] */
187:   29,           /* 0136 ^ word separator */
188:  -1,            /* 0137 _ illegal */
189:
190:  -1,            /* 0140 ` illegal */
191:   49,           /* 0141 a is A */
192:   16,           /* 0142 b is substitute blank */
193:   51,           /* 0143 c is C */
194:   52,           /* 0144 d is D */
195:   53,           /* 0145 e is E */
196:   54,           /* 0146 f is F */
197:   55,           /* 0147 g is G */
198:
```

```
199:    56,          /* 0150 h is H */
200:    57,          /* 0151 i is I */
201:    33,          /* 0152 j is J */
202:    34,          /* 0153 k is K */
203:    35,          /* 0154 l is L */
204:    36,          /* 0155 m is M */
205:    37,          /* 0156 n is N */
206:    38,          /* 0157 o is O */
207:
208:    39,          /* 0160 p is P */
209:    40,          /* 0161 q is Q */
210:    41,          /* 0162 r is R */
211:    18,          /* 0163 s is S */
212:    19,          /* 0164 t is T */
213:    20,          /* 0165 u is U */
214:    21,          /* 0166 v is V */
215:    22,          /* 0167 w is W */
216:
217:    23,          /* 0170 x is X */
218:    24,          /* 0171 y is Y */
219:    25,          /* 0172 z is Z */
220:    -1,          /* 0173 { illegal */
221:    26,          /* 0174 | substitute record mark */
222:    -1,          /* 0175 } illegal */
223:    -1,          /* 0176 ~ illegal */
224:    47,          /* 0177 □ delta */
225:
226:    -1,-1,-1,-1,-1,-1,-1,-1, /* 0200-0207 illegal */
227:    -1,-1,-1,-1,-1,-1,-1,-1, /* 0210-0217 illegal */
228:    -1,-1,-1,-1,-1,-1,-1,-1, /* 0220-0227 illegal */
229:    -1,-1,-1,-1,-1,-1,-1,-1, /* 0230-0237 illegal */
230:    -1,-1,-1,-1,-1,-1,-1,-1, /* 0240-0247 illegal */
231:    -1,-1,-1,-1,-1,-1,-1,-1, /* 0250-0257 illegal */
232:    -1,-1,-1,-1,-1,-1,-1,-1, /* 0260-0267 illegal */
233:    -1,-1,-1,-1,-1,-1,-1,-1, /* 0270-0277 illegal */
234:    -1,-1,-1,-1,-1,-1,-1,-1, /* 0300-0307 illegal */
235:
236:    -1,-1,-1,-1,-1,-1,      /* 0310-0315 illegal */
237:    63,          /* 0316 group mark */
238:    -1,          /* 0317 illegal */
239:
240:    -1,-1,-1,-1,-1,-1,-1, /* 0320-0326 illegal */
241:    31,          /* 0327 segment mark */
242:
243:    26,          /* 0330 record mark */
244:    -1,-1,-1,-1,-1,-1,-1, /* 0331-0337 illegal */
245:
246:    -1,-1,-1,-1,-1,-1,-1, /* 0340-0347 illegal */
247:    -1,-1,-1,-1,-1,-1,-1, /* 0350-0357 illegal */
248:    -1,-1,-1,-1,-1,-1,-1, /* 0360-0367 illegal */
249:    -1,-1,-1,          /* 0370-0372 illegal */
250:    15,          /* 373 radical */
251:    -1,-1,-1,-1};        /* 0374-0377 illegal */
252:
253: // Parity table. Encode the 64 BCD characters. Does not itself
254: // include the check bit, so the 2nd 64 are the same as the first
255: // 64. Does include the WM bit, so the next 128 entries are the
256: // opposite of the first 128.
257: //
258: //
259:
260: int parity_table[256] = {
261:     0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,
262:     1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,
263:
264:     0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,
```

```

265:     1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,
266:
267:     1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,
268:     0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,
269:
270:     1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,
271:     0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1
272: };
273:
274: // This table is similar to the one above, but is used for checking
275: // for correct odd parity. As a result, the 2nd 64 are the complement
276: // of the first 64 (because the Check bit is 0x40), and the 2nd 128
277: // are the complement of the 1st 128 (because the WM bit is 0x80).
278:
279: // Note that you can get a similar thing for even parity by using
280: // a logical ! operator on the table below.
281:
282: int odd_parity_table[256] = {
283:     0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,0,0,1,0,1,1,0,0,0,1,1,0,1,0,0,1,
284:     1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,
285:
286:     1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,
287:     0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,
288:
289:     1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,
290:     0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,
291:
292:     0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,
293:     1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,1,0,0,1
294: };
295:
296: // This table translates valide BCD numerics (odd parity, including
297: // check bit). It assumes that any WM bit has already been stripped.
298: // It also assumes that the character has the correct (ODD) parity!
299: // Note that the Two Out of Five code "0" bit is here coded as 16.
300:
301: int bcd_to_two_of_five_table[] = {
302:     0,17,18,0,20,0,0,12,24,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
303:     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
304:     0,0,0,3,0,5,6,0,0,9,10,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
305:     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
306: };
307:
308: int two_of_five_to_bin_table[] = {
309:     -1,-1,-1,3,-1,5,6,-1,-1,9,0,-1,7,-1,-1,-1,
310:     -1,1,2,-1,4,-1,-1,-1,8,-1,-1,-1,-1,-1,-1
311: };
312:
313: int bin_to_two_of_five_table[] = {
314:     10,17,18,3,20,5,6,12,24,9
315: };
316:
317: enum bcd_char_type bcd_char_type_table[64] = {
318:     BCD_NN, /* 0           - space */
319:     BCD_AN, /* 1           1   - 1 */
320:     BCD_AN, /* 2           2   - 2 */
321:     BCD_AN, /* 3           21  - 3 */
322:     BCD_AN, /* 4           4   - 4 */
323:     BCD_AN, /* 5           4 1  - 5 */
324:     BCD_AN, /* 6           42  - 6 */
325:     BCD_AN, /* 7           421 - 7 */
326:     BCD_AN, /* 8           8   - 8 */
327:     BCD_AN, /* 9           8 1  - 9 */
328:     BCD_AN, /* 10          8 2  - 0 */
329:     BCD_SC, /* 11          8 21 - number sign (#) or equal */
330:     BCD_SC, /* 12          84   - at sign @ or quote */

```

```
331:     BCD_SC, /* 13      84 1 - colon */
332:     BCD_SC, /* 14      842 - greater than */
333:     BCD_SC, /* 15      8421 - radical */
334:     BCD_NN, /* 16      A    - substitute blank */
335:     BCD_SC, /* 17      A    1 - slash */
336:     BCD_AN, /* 18      A    2 - S */
337:     BCD_AN, /* 19      A    21 - T */
338:     BCD_AN, /* 20      A    4 - U */
339:     BCD_AN, /* 21      A    4 1 - V */
340:     BCD_AN, /* 22      A    42 - W */
341:     BCD_AN, /* 23      A    421 - X */
342:     BCD_AN, /* 24      A8   - Y */
343:     BCD_AN, /* 25      A8   1 - Z */
344:     BCD_AN, /* 26      A8   2 - record mark */
345:     BCD_SC, /* 27      A8   21 - comma */
346:     BCD_SC, /* 28      A84  - percent % or paren */
347:     BCD_SC, /* 29      A84  1 - word separator */
348:     BCD_SC, /* 30      A842 - left oblique */
349:     BCD_SC, /* 31      A8421 - segment mark */
350:     BCD_NN, /* 32      B    - hyphen */
351:     BCD_AN, /* 33      B    1 - J */
352:     BCD_AN, /* 34      B    2 - K */
353:     BCD_AN, /* 35      B    21 - L */
354:     BCD_AN, /* 36      B    4 - M */
355:     BCD_AN, /* 37      B    4 1 - N */
356:     BCD_AN, /* 38      B    42 - O */
357:     BCD_AN, /* 39      B    421 - P */
358:     BCD_AN, /* 40      B    8 - Q */
359:     BCD_AN, /* 41      B    8 1 - R */
360:     BCD_AN, /* 42      B    8 2 - exclamation (-0) */
361:     BCD_SC, /* 43      B    8 21 - dollar sign */
362:     BCD_SC, /* 44      B    84 - asterisk */
363:     BCD_SC, /* 45      B    84 1 - right bracket */
364:     BCD_SC, /* 46      B    842 - semicolon */
365:     BCD_SC, /* 47      B    8421 - delta */
366:     BCD_NN, /* 48      BA   - ampersand or plus */
367:     BCD_AN, /* 49      BA   1 - A */
368:     BCD_AN, /* 50      BA   2 - B */
369:     BCD_AN, /* 51      BA   21 - C */
370:     BCD_AN, /* 52      BA   4 - D */
371:     BCD_AN, /* 53      BA   4 1 - E */
372:     BCD_AN, /* 54      BA   42 - F */
373:     BCD_AN, /* 55      BA   421 - G */
374:     BCD_AN, /* 56      BA8  - H */
375:     BCD_AN, /* 57      BA8  1 - I */
376:     BCD_AN, /* 58      BA8  2 - question mark */
377:     BCD_SC, /* 59      BA8  21 - period */
378:     BCD_SC, /* 60      BA84 - lozenge or paren */
379:     BCD_SC, /* 61      BA84 1 - left bracket */
380:     BCD_SC, /* 62      BA842 - less than */
381:     BCD_SC /* 63      BA8421 - group mark */;
```



```

67:     // Note that we don't do the address modification cycle yet, as we
68:     // have to store the result back where BAR points!
69:
70:     // The following looks complicated on the flow chart in the
71:     // IBM CE Instructional materials 1411 Processing Unit Instructions
72:     // on page 11. But, the end result is that if you have an EVEN number
73:     // of: -A, -B and SUBTRACT you do a TRUE ADD cycle, and ODD number and
74:     // you do a COMPLEMENT ADD cycle
75:
76:     if(ScanRing -> State() == SCAN_1) {           // Is 1st scan ring on?
77:         if(SubScanRing -> State() == SUB_SCAN_U) { // Units latch?
78:             BComplement -> Reset();                // Set True Add B latch
79:             i =
80:                 (A_Reg -> Get().IsMinus()) +
81:                 (B_Reg -> Get().IsMinus()) +
82:                 (op_bin == OP_SUBTRACT);
83:             if(i & 1) {
84:                 AComplement -> Set();                  // Odd #: complement add
85:                 CarryIn -> Set();
86:             }
87:             else {
88:                 AComplement -> Reset();                // Even #: true add
89:                 CarryIn -> Reset();
90:             }
91:
92:             adder_a = A_Reg -> Get();
93:
94:         } // End, units
95:         else {
96:             if(SubScanRing -> State() == SUB_SCAN_E) { // Extension latch?
97:                 adder_a = BCD_0;                         // Yes, gate '0'
98:                                         // (if a compl then 9)
99:             }
100:            else {
101:                adder_a = A_Reg -> Get();              // No, use A data
102:            }
103:        }
104:    } // End, SCAN 1
105:
106:    else {
107:        assert(ScanRing -> State() == SCAN_3);      // Else must be 3
108:        if(SubScanRing -> State() == SUB_SCAN_U) {    // Units?
109:            b_temp = B_Reg -> Get();                  // Yes. Invert sign
110:            b_temp = (b_temp & ~BIT_ZONE) |
111:                      (b_temp.IsMinus() ? (BITA | BITB) : BITB);
112:            b_temp.SetOddParity();
113:            B_Reg -> Set(b_temp);
114:            AComplement -> Reset();                  // Reset compl add
115:        }
116:        adder_a = BCD_0;
117:    }
118:
119: // Run the right stuff thru the adder, and then thru the Assembly Channel
120:
121: Adder(addr_a,AComplement -> State(),
122:       B_Reg -> Get(),BComplement -> State());
123:
124: if(ZeroBalance -> State() &&
125:     (AdderResult & BIT_NUM) != (BCD_0 & ~BITC)) {
126:     ZeroBalance -> Reset();
127: }
128:
129: AssemblyChannel -> Select(
130:     AssemblyChannel -> AsmChannelWMB,
131:     AssemblyChannel -> AsmChannelZonesB,
132:     false,

```

```
133:         AssemblyChannel -> AsmChannelSignNone,
134:         AssemblyChannel -> AsmChannelNumAdder
135:     );
136:
137:     if(B_Reg -> Get().TestWM()) { // B Channel WM?
138:
139:         if(ScanRing -> State() == SCAN_3) { // 3rd scan latch?
140:             Store(AssemblyChannel -> Select()); // Store the last char
141:             Cycle(); // Modify B Addr -1
142:             IRingControl = true; // Done!
143:             return;
144:         }
145:
146:         if(AComplement -> State()) { // Complement add?
147:
148:             if(CarryOut -> State()) { // Carry too?
149:                 Store(AssemblyChannel -> Select()); // Store the last char
150:                 Cycle(); // Modify B addr -1
151:                 IRingControl = true; // Done! (no sign sw)
152:                 return;
153:             }
154:
155:             // What I *think* the following does is to reverse the B sign
156:             // where D_AR points, because SUB_SCAN_U causes the CPU to use
157:             // the DAR for the storage cycle, which starts a recomplement
158:             // cycle wherein the entire B field is recomplemented. This
159:             // can only happen if we are doing a complement add,
160:             // with no carry out.
161:
162:             SubScanRing -> Set(SUB_SCAN_U); // Set units latch
163:             ScanRing -> Set(SCAN_3); // Set 3rd scan latch
164:             BComplement -> Set(); // Set B compl add
165:             CarryIn -> Set(); // Set carry
166:             Store(AssemblyChannel -> Select()); // Store the last char
167:             Cycle(); // Finish the B cycle
168:             cycle_type = CYCLE_B; // Re-enter at B cycle
169:             return;
170:         }
171:
172:     else { // True add
173:         if(CarryOut -> State()) { // Set overflow on carry
174:             Overflow -> Set();
175:         }
176:         Store(AssemblyChannel -> Select()); // Store the last char
177:         Cycle(); // Modify B addr -1
178:         IRingControl = true; // Done!
179:         return;
180:     }
181: } // End B Ch WM
182:
183: else { // No B Ch WM
184:     CarryIn -> Set(CarryOut -> State()); // Set carry
185:     Store(AssemblyChannel -> Select()); // Store result char
186:     Cycle(); // Modify B address -1
187:     if(ScanRing -> State() == SCAN_1) { // Still 1st scan?
188:         if(A_Reg -> Get().TestWM()) { // A Ch WM?
189:             SubScanRing -> Set(SUB_SCAN_E); // Yes -- set extension
190:             cycle_type = CYCLE_B; // No more A cycles!
191:             return; // continue on
192:         }
193:         else { // No A CH WM
194:             SubScanRing -> Set(SUB_SCAN_B); // Set to body
195:             cycle_type = CYCLE_A; // A cycle next
196:             return;
197:         }
198:     }
}
```

```
199:     else {
200:         assert(ScanRing -> State() == SCAN_3);           // Must be!
201:         SubScanRing -> Set(SUB_SCAN_E);                 // Still in extension
202:         cycle_type = CYCLE_B;                           // Which means no A cyc
203:         return;
204:     }
205: }
206: }
207:
208: void T1410CPU::InstructionZeroArith()
209: {
210:
211:     // Variable to maintain state between cycles
212:
213:     static int cycle_type;
214:     static int op_bin;
215:
216:     BCD a_temp,a_hold;
217:
218:     // If this is the first time in, set things up
219:
220:     if(LastInstructionReadout) {
221:         SubScanRing -> Set(SUB_SCAN_U);                  // Units
222:         ScanRing -> Set(SCAN_1);                      // Scan 1 (-1 mod)
223:
224:         AComplement -> Reset();
225:         BComplement -> Reset();
226:         CarryIn -> Reset();
227:         CarryOut -> Reset();
228:         ZeroBalance -> Set();                          // Assume 0 at start
229:
230:         cycle_type = CYCLE_A;                           // Set for 1st A cycle
231:         op_bin = Op_Reg -> Get().ToInt() & 0x3f;
232:         LastInstructionReadout = false;
233:     }
234:
235:     // Initial A cycle
236:
237:     if(cycle_type == CYCLE_A) {
238:         CycleRing -> Set(CYCLE_A);                   // Take an A Cycle
239:         *STAR = *A_AR;
240:         Readout();                                     // RO A field char.
241:         Cycle();                                       // Set AAR with mod
242:
243:         // Units or body latch Regen happens by itself in simulator
244:
245:         cycle_type = CYCLE_B;                         // Set for B cycle
246:         return;
247:     }
248:
249:     assert(cycle_type == CYCLE_B);                   // Has to be B cycle
250:
251:     CycleRing -> Set(CYCLE_B);                   // Take B Cycle
252:     if(SubScanRing -> State() == SUB_SCAN_U) {      // Read B units via DAR
253:         *STAR = *D_AR;
254:     }
255:     else {
256:         *STAR = *B_AR;
257:     }
258:     Readout();                                     // Ro B Field Char
259:
260:     // If we are in the body/extension, we just insert 0's on the
261:     // A channel. Otherwise, we normalize the A field sign
262:
263:     a_temp = A_Reg -> Get();                      // Save to restore ...
264:     a_hold = a_temp;
```



```

331:     }
332: }
333:
334: // Multiply instruction execution routine
335:
336: void T1410CPU::InstructionMultiply()
337: {
338:     static int cycle_type;
339:     static int cycle_subtype;
340:     static bool MultiplierDigitLatch;
341:
342:     BCD adder_a,b_temp;
343:
344:     if(LastInstructionReadout) {
345:         SubScanRing -> Set(SUB_SCAN_U);           // Set the Units latch
346:         ScanRing -> Set(SCAN_1);                // Set Scan to 1
347:         BComplement -> Reset();                 // True Add B
348:         AComplement -> Reset();                 // True Add A
349:         ZeroBalance -> Set();                  // Start as 0
350:         CarryIn -> Reset();                   // So single cycle
351:         CarryOut -> Reset();                  // looks good
352:         MultiplierDigitLatch = false;          // MDL latch reset
353:         SignLatch = false;                     // Start out as +
354:         cycle_type = CYCLE_A;                  // Start with A cycle
355:         cycle_subtype = 0;                      // No subtype
356:     }
357:
358:     if(cycle_type == CYCLE_A && cycle_subtype == 0) {
359:         CycleRing -> Set(CYCLE_A);           // Set for an A cycle
360:         if(SubScanRing -> State() == SUB_SCAN_U ||
361:             SubScanRing -> State() == SUB_SCAN_E) {
362:             *STAR = *C_AR;                    // A units, extension via CAR
363:         }
364:         else {
365:             assert(SubScanRing -> State() == SUB_SCAN_B);
366:             *STAR = *A_AR;                  // A body via AAR
367:         }
368:         Readout();
369:         Cycle();
370:         assert(ScanRing -> State() == SCAN_1); // Regen 1st Scan
371:         cycle_type = CYCLE_B;                // Next cycle: B
372:         cycle_subtype = 0;
373:         return;
374:     } // End Cycle A
375:
376:     if(cycle_type == CYCLE_B && cycle_subtype == 0) {
377:
378:         CycleRing -> Set(CYCLE_B);           // Set for a B cycle
379:         if(SubScanRing -> State() == SUB_SCAN_U) {
380:             *STAR = *D_AR;                  // Read B units - DAR
381:         }
382:         else {
383:             *STAR = *B_AR;                  // Otherwise, use BAR
384:         }
385:         Readout();
386:
387:         if(SubScanRing -> State() == SUB_SCAN_U ||
388:             SubScanRing -> State() == SUB_SCAN_B ||
389:             SubScanRing -> State() == SUB_SCAN_E) {
390:
391:             Adder(BCD_0,AComplement->State(),
392:                   BCD_0,BComplement->State());
393:             AssemblyChannel -> Select(
394:                 AssemblyChannel -> AsmChannelWMB,
395:                 AssemblyChannel -> AsmChannelZonesNone,
396:                 false,

```

```

397:             AssemblyChannel -> AsmChannelSignNone,
398:             AssemblyChannel -> AsmChannelNumAdder);
399:             Store(AssemblyChannel -> Select()); // Store the 0
400:             // Modify B address by -1
401:             assert(!AComplement -> State())); // regen True add
402:
403:             if(SubScanRing -> State() == SUB_SCAN_E) { // Extension
404:                 CarryIn -> Reset(); // No carry
405:                 SubScanRing -> Set(SUB_SCAN_MQ); // Set MQ latch
406:                 assert(ScanRing -> State() == SCAN_1); // Regen 1st scan
407:                 cycle_type = CYCLE_B; // B Cycle next
408:                 cycle_subtype = 0;
409:                 return;
410:             } // Extension
411:
412:             assert(SubScanRing -> State() == SUB_SCAN_U || // Units or Body
413:                   SubScanRing -> State() == SUB_SCAN_B);
414:
415:             if(A_Reg -> Get().TestWM()) { // A Channel WM?
416:                 SubScanRing -> Set(SUB_SCAN_E); // Set Extension latch
417:                 ScanRing -> Set(SCAN_N); // Set No Scan
418:                 cycle_type = CYCLE_C; // C Cycle next
419:                 cycle_subtype = 0;
420:                 return;
421:             }
422:             else {
423:                 SubScanRing -> Set(SUB_SCAN_B); // Set Body latch
424:                 assert(ScanRing -> State() == SCAN_1); // Regen 1st scan
425:                 cycle_type = CYCLE_A; // A Cycle next
426:                 cycle_subtype = 0;
427:                 return;
428:             } // End: Units or body
429:
430:         } // End: Units, body or Extension
431:
432:     else {
433:         assert(SubScanRing -> State() == SUB_SCAN_MQ); // MQ
434:
435:         // As with add and subtract, the following is complicated on
436:         // the flow chart, but all it is really doing is checking to
437:         // see if the number of "-" values is even or odd
438:
439:         if((A_Reg -> Get().IsMinus()) + (B_Reg -> Get().IsMinus()) & 1) {
440:             CPU -> SignLatch = true; // Odd: Set - sign
441:         }
442:         else {
443:             CPU -> SignLatch = false;
444:         }
445:
446:         DEBUG("Multiply, B Cycle, Sign is %d .",CPU -> SignLatch);
447:
448:         adder_a = BCD_9; // 9 on adder A ch.
449:         b_temp = B_Reg -> Get(); // Analyze B ch char
450:         b_temp = b_temp & BIT_NUM; // Look at just numerics
451:         if(b_temp == (BCD_0 & BIT_NUM)) { // Store 0 in B Field
452:             Store(AssemblyChannel -> Select (
453:                 AssemblyChannel -> AsmChannelWMB,
454:                 AssemblyChannel -> AsmChannelZonesNone,
455:                 false,
456:                 AssemblyChannel -> AsmChannelSignNone,
457:                 AssemblyChannel -> AsmChannelNumZero));
458:             Cycle();
459:             AComplement -> Reset(); // Set True add latch
460:             CarryIn -> Reset(); // Set No Carry
461:             if(B_Reg -> Get().TestWM()) {
462:                 ScanRing -> Set(SCAN_2); // Set 2nd Scan

```

```

463:         }
464:     else {
465:         assert(ScanRing -> State() == SCAN_1); // Regen 1st scan
466:     }
467:   }
468: else if(b_temp.ToInt() > 4) { // 5 ~ 9
469:   Store( AssemblyChannel -> Select ( // Store B numerics
470:           AssemblyChannel -> AsmChannelWMB,
471:           AssemblyChannel -> AsmChannelZonesNone,
472:           false,
473:           AssemblyChannel -> AsmChannelSignNone,
474:           AssemblyChannel -> AsmChannelNumB) );
475:   Cycle();
476:   AComplement -> Set(); // Set Complement
477:   CarryIn -> Set(); // Set Carry
478:   ScanRing -> Set(SCAN_N); // Set No Scan
479: }
480:
481: else {
482:   Adder(adder_a,AComplement->State(),b_temp,BComplement->State());
483:   Store( AssemblyChannel -> Select ( // Store adder output
484:           AssemblyChannel -> AsmChannelWMB,
485:           AssemblyChannel -> AsmChannelZonesNone,
486:           false,
487:           AssemblyChannel -> AsmChannelSignNone,
488:           AssemblyChannel -> AsmChannelNumAdder) );
489:   Cycle();
490:   AComplement -> Reset(); // Set True Latch
491:   CarryIn -> Reset(); // No Carry
492:   ScanRing -> Set(SCAN_N); // Set No Scan
493: }
494:
495: cycle_type = CYCLE_D;
496: cycle_subtype = 0;
497: return;
498:
499: } // MQ
500:
501: } // End B Cycle, subtype 0
502:
503: if(cycle_type == CYCLE_C) {
504:   CycleRing -> Set(CYCLE_C); // C Cycle
505:   *STAR = *C_AR; // Read out units pos
506:   Readout(); // Read storage
507:   Cycle(); // C is not modified
508:   ScanRing -> Set(SCAN_1);
509:   cycle_type = CYCLE_B; // B Cycle next time
510:   cycle_subtype = 0;
511:   return;
512: } // End C Cycle
513:
514: if(cycle_type == CYCLE_D) {
515:   CycleRing -> Set(CYCLE_D); // D Cycle
516:   *STAR = *D_AR;
517:   Readout();
518:   if(ScanRing -> State() != SCAN_3) { // N, 1st or 2nd scan
519:     Store( AssemblyChannel -> Select(
520:             AssemblyChannel -> AsmChannelWMB,
521:             AssemblyChannel -> AsmChannelZonesNone,
522:             false,
523:             AssemblyChannel -> AsmChannelSignLatch,
524:             AssemblyChannel -> AsmChannelNumZero) );
525:   }
526:   // If 3rd Scan memory is regened (no store in emulator)
527:
528:   Cycle();

```

```

529:     CarryIn -> Set(AComplement -> State());           // Set carry state
530:
531:
532:     if(ScanRing -> State() == SCAN_2) {                  // Done if 2nd Scan
533:         IRingControl = true;
534:         return;
535:     }
536:
537:     if(ScanRing -> State() == SCAN_N || MultiplierDigitLatch) {
538:         SubScanRing -> Set(SUB_SCAN_U);                   // Set Units latch
539:         ScanRing -> Set(SCAN_3);                         // Set 3rd scan
540:         BComplement -> Reset();                          // Set True Add B
541:         // AComplement regen'd
542:         cycle_type = CYCLE_A;                           // A Cycle next
543:         cycle_subtype = 1;                             // 2nd kind of A cycle
544:         return;
545:     }
546:
547:     // 1st or 3rd scan, MDL latch reset...
548:
549:     SubScanRing -> Set(SUB_SCAN_MQ);                 // Set MQ latch
550:     ScanRing -> Set(SCAN_3);                         // Set 3rd scan
551:     BComplement -> Reset();                          // True add B latch
552:     // AComplement regen'd
553:     cycle_type = CYCLE_B;                           // B Cycle next
554:     cycle_subtype = 1;                             // 2nd kind of B cycle
555:     return;
556: } // End D Cycle
557:
558: if(cycle_type == CYCLE_A && cycle_subtype == 1) {          // A Cycle
559:     CycleRing -> Set(CYCLE_A);
560:     if(SubScanRing -> State() == SUB_SCAN_U ||           // Units or Extension?
561:         SubScanRing -> State() == SUB_SCAN_E) {           // Yes, use CAR for address
562:         *STAR = *C_AR;
563:     }
564:     else {
565:         *STAR = *A_AR;
566:     }
567:     Readout();                                         // Read out A field
568:     Cycle();
569:     assert(ScanRing -> State() == SCAN_3);             // Regen 3rd scan
570:     BComplement -> Reset();                          // True add B latch
571:     // A Complement regen'd
572:     cycle_type = CYCLE_B;                           // B Cycle next
573:     cycle_subtype = 1;                             // 2nd kind of B cycle
574:     return;
575: }
576:
577: if(cycle_type == CYCLE_B && cycle_subtype == 1) {          // B Cycle
578:
579:     CycleRing -> Set(CYCLE_B);
580:     if(SubScanRing -> State() == SUB_SCAN_U) {           // Read Units using DAR
581:         *STAR = *D_AR;
582:     }
583:     else {
584:         *STAR = *B_AR;
585:     }
586:     Readout();
587:
588:     if(SubScanRing -> State() == SUB_SCAN_MQ) {           // Insert 0 on A
589:         if(AComplement -> State()) {
590:             adder_a = BCD_0;                                // Carry is also set
591:             q = 1;                                         // Will add 1 to B
592:         }
593:     }
594: }

```

```

595:         adder_a = BCD_9;                                // Insert 9 on A
596:                                                 // Will decrement B
597:     }
598:     b_temp = B_Reg -> Get();                         // Get B data
599:     b_temp = b_temp & BIT_NUM;                        // Numeric bits
600:
601:     if(b_temp == (BCD_0 & BIT_NUM) && !(AComplement -> State()) ) {
602:         Store( AssemblyChannel -> Select (           // Store 0, no zones
603:             AssemblyChannel -> AsmChannelWMB,
604:             AssemblyChannel -> AsmChannelZonesNone,
605:             false,
606:             AssemblyChannel -> AsmChannelSignNone,
607:             AssemblyChannel -> AsmChannelNumZero) );
608:         Cycle();
609:         assert(! (AComplement -> State()));          // True Add (was already)
610:         CarryIn -> Reset();                           // Reset Carry
611:         if(B_Reg -> Get().TestWM()) {                  // B WM?
612:             IRingControl = true;                         // Yes. All done.
613:             return;
614:         }
615:         assert(ScanRing -> State() == SCAN_3);        // Regen 3rd Scan latch
616:         assert(! (BComplement -> State()));           // Regen B True add
617:         cycle_type = CYCLE_D;                          // Next cycle D
618:         cycle_subtype = 0;
619:         return;
620:     } // End B is Zero, True Add
621:
622:     if(AComplement -> State() &&
623:         (b_temp == (BCD_0 & BIT_NUM) || b_temp.ToInt() < 5)) {
624:         Store ( AssemblyChannel -> Select (           // Store B (no zones)
625:             AssemblyChannel -> AsmChannelWMB,
626:             AssemblyChannel -> AsmChannelZonesNone,
627:             false,
628:             AssemblyChannel -> AsmChannelSignNone,
629:             AssemblyChannel -> AsmChannelNumB) );
630:         Cycle();
631:         AComplement -> Reset();                      // Set True Add latch
632:         CarryIn -> Reset();                          // Set Units latch
633:         SubScanRing -> Set(SUB_SCAN_U);              // Set Units latch
634:         assert(ScanRing -> State() == SCAN_3);        // Regen 3rd scan
635:         BComplement -> Reset();                      // Set True Add B latch
636:         cycle_type = CYCLE_A;                         // Next Cycle: A
637:         cycle_subtype = 1;                            // second kind of A
638:         return;
639:     } // End B is 0-4, Complement Add
640:
641:     if(! (AComplement -> State()) && b_temp.ToInt() < 5) {
642:         Adder(addr_a,AComplement -> State(),
643:                B_Reg -> Get(),BComplement -> State());
644:         Store ( AssemblyChannel -> Select (           // Store adder, no zones
645:             AssemblyChannel -> AsmChannelWMB,
646:             AssemblyChannel -> AsmChannelZonesNone,
647:             false,
648:             AssemblyChannel -> AsmChannelSignNone,
649:             AssemblyChannel -> AsmChannelNumAdder) );
650:         Cycle();
651:         AComplement -> Reset();                      // Set True Add latch
652:         CarryIn -> Reset();                          // Set no carry
653:         SubScanRing -> Set(SUB_SCAN_U);              // Set units latch
654:         assert(ScanRing -> State() == SCAN_3);        // Regen 3rd scan
655:         BComplement -> Reset();                      // True add B
656:         cycle_type = CYCLE_A;                         // Next cycle A
657:         cycle_subtype = 1;                            // second kind of A
658:         return;
659:     } // End B is 1-4, True Add
660:
```

```

661:         assert(b_temp != (BCD_0 & BIT_NUM) && b_temp.ToInt() >= 5);
662:
663:         if(b_temp.ToInt() < 9 && AComplement -> State()) {
664:             Adder(addr_a,AComplement -> State(),
665:                   B_Reg -> Get(),BComplement -> State());
666:             Store( AssemblyChannel -> Select (           // Store adder, no zones
667:                                         AssemblyChannel -> AsmChannelWMB,
668:                                         AssemblyChannel -> AsmChannelZonesNone,
669:                                         false,
670:                                         AssemblyChannel -> AsmChannelSignNone,
671:                                         AssemblyChannel -> AsmChannelNumAdder) );
672:             Cycle();
673:             AComplement -> Set();                         // Set Complement latch
674:             CarryIn -> Set();                           // Set carry latch
675:             SubScanRing -> Set(SUB_SCAN_U);            // Set Units latch
676:             assert(ScanRing -> State() == SCAN_3);      // Regen 3rd scan
677:             BComplement -> Reset();                     // True add B
678:             cycle_type = CYCLE_A;                      // Next Cycle A
679:             cycle_subtype = 1;                          // second kind of A
680:             return;
681:         }
682:
683:         if(!(AComplement -> State())) {               // True latch on?
684:             Store( AssemblyChannel -> Select (           // Store B, no zones
685:                                         AssemblyChannel -> AsmChannelWMB,
686:                                         AssemblyChannel -> AsmChannelZonesNone,
687:                                         false,
688:                                         AssemblyChannel -> AsmChannelSignNone,
689:                                         AssemblyChannel -> AsmChannelNumB) );
690:             Cycle();
691:             AComplement -> Set();                         // Set Complement latch
692:             CarryIn -> Set();                           // Set carry latch
693:             SubScanRing -> Set(SUB_SCAN_U);            // Set Units latch
694:             assert(ScanRing -> State() == SCAN_3);      // Regen 3rd scan
695:             BComplement -> Reset();                     // True add B
696:             cycle_type = CYCLE_A;                      // Next Cycle A
697:             cycle_subtype = 1;                          // second kind of A
698:             return;
699:         }
700:
701:         assert(b_temp.ToInt() == 9 && AComplement -> State());
702:
703:         Store(AssemblyChannel -> Select (           // Store 0 or B, no zones
704:                                         AssemblyChannel -> AsmChannelWMB,
705:                                         AssemblyChannel -> AsmChannelZonesNone,
706:                                         false,
707:                                         AssemblyChannel -> AsmChannelSignNone,
708:                                         (ZeroBalance -> State() ?
709:                                         AssemblyChannel -> AsmChannelNumZero :
710:                                         AssemblyChannel -> AsmChannelNumB) );
711:         Cycle();
712:         if(B_Reg -> Get().TestWM()) {                  // B WM?
713:             AComplement -> Reset();                     // Set True Add latch
714:             MultiplierDigitLatch = true;                // Set MDL latch
715:         }
716:         else {
717:             AComplement -> Set();                      // No. Set Complement
718:         }
719:         assert(ScanRing -> State() == SCAN_3);        // Regen 3rd scan
720:         BComplement -> Reset();                      // True Add B
721:         cycle_type = CYCLE_D;                        // D Cycle Next
722:         cycle_subtype = 0;
723:         return;
724:     } // End: MQ
725:
726:     if(SubScanRing -> State() == SUB_SCAN_E) {

```

```

727: if(AComplement -> State()) {
728:   adder_a = BCD_9;
729:   +
730: } else {
731:   adder_a = BCD_0;
732:   +
733: }
734: else {
735:   adder_a = A_Reg -> Get();
736: }
737:
738: Adder(addr_a, AComplement -> State(),
739:       B_Reg -> Get(), BComplement -> State());
740:
741: Store ( AssemblyChannel -> Select (           // Adder, B Zones to Asm
742:         AssemblyChannel -> AsmChannelWMB,
743:         AssemblyChannel -> AsmChannelZonesB,
744:         false,
745:         AssemblyChannel -> AsmChannelSignNone,
746:         AssemblyChannel -> AsmChannelNumAdder) );
747: Cycle();
748:
749: if((AdderResult & BIT_NUM) == (BCD_0, & ~BITC)) { // Check Zero Balance
750:   ZeroBalance -> Reset();
751: }
752:
753: if(SubScanRing -> State() == SUB_SCAN_E) {      // Extension?
754:   CarryIn -> Set(AComplement -> State());      // Yes. Possibly set carry
755:   if(MultiplierDigitLatch) {                      // MDL ?
756:     IRingControl = true;                          // Yes - Done
757:     return;
758:   }
759:   SubScanRing -> Set(SUB_SCAN_MQ);               // No - set MQ latch
760:   assert(ScanRing -> State() == SCAN_3);          // Regen 3rd scan
761:   BComplement -> Reset();                         // True Add B
762:   cycle_type = CYCLE_B;                           // Take another B cycle
763:   cycle_subtype = 1;                             // also subtype 1
764:   return;
765: }
766:
767: assert(SubScanRing -> State() == SUB_SCAN_U || // Must be Units or Body
768:        SubScanRing -> State() == SUB_SCAN_B);      //
769:
770: CarryIn -> Set(CarryOut -> State());          // Forward adder carry
771:
772: if(A_Reg -> Get().TestWM()) {                   // A Channel WM ?
773:   SubScanRing -> Set(SUB_SCAN_E);              // Yes. Set Extension
774:   assert(ScanRing -> State() == SCAN_3);        // Regen 3rd scan
775:   cycle_type = CYCLE_B;                         // Take another B Cycle
776:   cycle_subtype = 1;                            // Also subtype 1
777:   return;
778: }
779: else {
780:   SubScanRing -> Set(SUB_SCAN_B);              // No WM. Body.
781:   assert(ScanRing -> State() == SCAN_3);        // Regen 3rd scan
782:   BComplement -> Reset();                       // True Add B
783:   cycle_type = CYCLE_A;                         // Take an A cycle next
784:   cycle_subtype = 1;                            // Subtype 1
785:   return;
786: }
787: } // End: B Cycle, subtype 1
788:
789: }
790:
791:
792:

```

complement  
helpers  
later

!=

793:

794:

795:

```

790:
791:
792: // Divide Instruction execute routine
793:
794: void T1410CPU::InstructionDivide()
795: {
796:
797:     // Divide is using a new approach to state, so that when doing
798:     // storage cycles, the storage print-out and indicators are the
799:     // state of the machine at the end of the instruction. (The other
800:     // instruction routines currently show scan and sub_scan for the
801:     // *next* cycle, which confuses debugging.
802:
803:     static MultiplyDivideLastLatch;
804:
805:     static struct {
806:         char cycle;
807:         char subcycle;
808:         char scan;
809:         char subscan;
810:     } next;
811:
812:     if>LastInstructionReadout) {
813:         SubScanRing -> Set(SUB_SCAN_U);
814:         next.subscan = SUB_SCAN_U;
815:         ScanRing -> Set(SCAN_1);
816:         next.scan = SCAN_1;
817:         BComplement -> Reset();           // True Add B
818:         AComplement -> Set();            // Complement add A
819:         CarryIn -> Set();               // NOT reset!
820:         CarryOut -> Set();              // NOT reset!
821:         MultiplyDivideLastLatch = false;
822:         SignLatch = false;
823:         SetBorrowVar;      NOT in divide          // Entering A cycle
824:         next.cycle = CYCLE_A;           // First kind
825:         next.subcycle = 0;
826:     }
827:
828:     // Do common items for all cycle types
829:
830:     CycleRing -> Set(next.cycle);
831:     ScanRing -> Set(next.scan);
832:     SubScanRing -> Set(next.subscan);
833:
834:     // The "if" statements use the next.variables, for easy of coding..
835:
836:     if(next.cycle == CYCLE_A && next.subcycle == 0) {
837:         assert(SubScanRing -> State() == SUB_SCAN_U);    // Use CAR to read units
838:         *STAR = *C_AR;                                     // Read into B register
839:         Readout();                                         // A Cycle copies to A Reg
840:         Cycle();                                           // And selects A Reg in
841:                                                       // A Channel.
842:
843:         next.cycle = CYCLE_B;
844:         next.subcycle = 0;
845:         return;
846:     }
847:
848:     if(next.cycle == CYCLE_B && next.subcycle == 0) {
849:
850:         if(SubScanRing -> State() == SUB_SCAN_U) {        // Units ?
851:             *STAR = *D_AR;                                  // Use DAR to read units
852:             Readout();                                     // True add cycle?
853:             if(!AComplement -> State()) &&           // Yes. B Bit set?
854:                 ((B_Reg -> Get()) & BITB) != 0 ) {       // Yes. Set MDL.
855:                 MultiplyDivideLastLatch = true;

```

```

856:             if((A_Reg -> Get()).IsMinus() !=           // Different signs?
857:                 (B_Reg -> Get()).IsMinus()) ) {
858:                 CPU -> SignLatch = true;                // Yes. Set minus latch
859:             }
860:             else {
861:                 CPU -> SignLatch = false;
862:             }
863:         }
864:     }
865:     else {                                         // Not units...
866:         *STAR = *B_AR;                            // Use BAR for readout
867:         Readout();
868:     } // End: Units
869:
870:     if(SubScanRing -> State() == SUB_SCAN_U ||      // Units or Body?
871:         SubScanRing -> State() == SUB_SCAN_B) {
872:         Adder(A_Reg -> Get(), AComplement -> State(), // Add A, B
873:             B_Reg -> Get(), BComplement -> State());
874:         Store( AssemblyChannel -> Select (          // And store in *BAR
875:             AssemblyChannel -> AsmChannelWMB,      B
876:             AssemblyChannel -> AsmChannelZonesNone,
877:             false,
878:             AssemblyChannel -> AsmChannelSignNone, // Sign handled later
879:             AssemblyChannel -> AsmChannelNumAdder) );
880:         Cycle();
881:         CarryIn -> Set(CarryOut -> State());        // Use adder to set carry
882:         if(A_Reg -> Get().TestWM()) {                  // A Channel WM?
883:             next.subscan = SUB_SCAN_E;                  // Yes --> Extension
884:             return;
885:         }
886:         else {
887:             next.subscan = SUB_SCAN_B;                  // No. Set Body
888:             next.cycle = CYCLE_A;                      // Next cycle A
889:             next.subcycle = 1;                          // Second kind
890:             return;
891:         }
892:     } // End: Units, Body
893:
894:     if(SubScanRing -> State() == SUB_SCAN_E) {        // Extension?
895:
896:         // Add a 0 (or 9 if complement is on) to B, and store
897:         Adder(BCD_0, AComplement -> State(),
898:             B_Reg -> Get(), BComplement -> State());
899:         Store( AssemblyChannel -> Select (
900:             AssemblyChannel -> AsmChannelWMB,      B
901:             AssemblyChannel -> AsmChannelZonesNone,
902:             false,
903:             AssemblyChannel -> AsmChannelSignNone,
904:             AssemblyChannel -> AsmChannelNumAdder) );
905:         Cycle();
906:         CarryIn -> Set(CarryOut -> State());        // Set carry from adder
907:
908:         if(AComplement -> State()) {                  // Complement cycle?
909:             if(CarryOut -> State()) {                  // Adder carry ?
910:                 AComplement -> Set();                  // Yes. Complement next
911:                 next.subscan = SUB_SCAN_MQ;            // Set MQ latch
912:                 next.cycle = CYCLE_B;                  // B Cycle next
913:                 next.subcycle = 0;                      // First kind.
914:                 return;
915:             }
916:             else {                                     // No adder carry.
917:                 AComplement -> Reset();              // True add next
918:                 next.subscan = SUB_SCAN_U;            // Set Units latch
919:                 next.scan = SCAN_3;                  // Set 3rd scan
920:                 next.cycle = CYCLE_A;                  // A Cycle next
921:                 next.subcycle = 1;                      // Second kind

```

```

922:             return;
923:         }
924:     } // End: Complement
925:
926:     if(MultiplyDivideLastLatch) { // Is MDL latch on?
927:         next.subscan = SUB_SCAN_MQ; // Set MQ Latch
928:         next.scan = SCAN_3; // Set 3rd scan
929:         AComplement -> Reset(); // True add next
930:         next.cycle = CYCLE_B; // B Cycle next
931:         next.subcycle = 0; // First kind
932:         return;
933:     }
934:     else { // No MDL latch
935:         next.scan = SCAN_2; // Set 2nd scan
936:         next.cycle = CYCLE_D; // D Cycle next
937:         next.subcycle = 0; // First kind
938:         return;
939:     }
940:
941: } // End: Extension
942:
943: assert(SubScanRing -> State() == SUB_SCAN_MQ); // Must be MQ
944:
945: if(AComplement -> State()) { // Complement ?
946:     assert(CarryIn -> State()); // Carry s/b on too
947:     Adder(BCD_0, false, B_Reg -> Get(), BComplement -> State()); // ++B
948:     Store(AssemblyChannel -> Select(
949:         AssemblyChannel -> AsmChannelWMB, B
950:         AssemblyChannel -> AsmChannelZonesNone,
951:         false,
952:         AssemblyChannel -> AsmChannelSignNone,
953:         AssemblyChannel -> AsmChannelNumAdder));
954:     Cycle();
955:     if(CarryOut -> State()) { // Adder carry?
956:         DivideOverflow -> Set(); // Yes. Set overflow!
957:         IRingControl = true; // And be done.
958:         return;
959:     }
960:     CarryIn -> Set(); // No. But set carry now
961:     next.subscan = SUB_SCAN_U; // Set units latch
962:     next.scan = SCAN_3; // Set 3rd scan
963:     AComplement -> Set(); // Complement Add next
964:     next.cycle = CYCLE_A; // A cycle next
965:     next.subcycle = 1; // Second kind.
966:     return;
967: } // End: MQ, Complement
968:
969: assert(SubScanRing -> State() == SUB_SCAN_MQ && MultiplyDivideLastLatch);
970:
971: assert(!(AComplement -> State())); // Regen True Add
972: Adder(BCD_9, false, B_Reg -> Get(), BComplement -> State()); // --B
973: Store(AssemblyChannel -> Select(
974:     AssemblyChannel -> AsmChannelWMB,
975:     AssemblyChannel -> AsmChannelZonesNone,
976:     false,
977:     AssemblyChannel -> AsmChannelSignLatch,
978:     AssemblyChannel -> AsmChannelNumAdder));
979: Cycle(); // And be done.
980: IRingControl = true;
981: return;
982:
983: } // End: B Cycle, subtype 0
984:
985: if(next.cycle == CYCLE_D) { // D reg to RO B field
986:     *STAR = *D_AR;
987:     Readout();

```





```
67: // Note that we don't do the address modification cycle yet, as we
68: // have to store the result back where BAR points!
69:
70: // The following looks complicated on the flow chart in the
71: // IBM CE Instructional materials 1411 Processing Unit Instructions
72: // on page 11. But, the end result is that if you have an EVEN number
73: // of: -A, -B and SUBTRACT you do a TRUE ADD cycle, and ODD number and
74: // you do a COMPLEMENT ADD cycle
75:
76: if(ScanRing -> State() == SCAN_1) { // Is 1st scan ring on?
77:     if(SubScanRing -> State() == SUB_SCAN_U) { // Units latch?
78:         BComplement -> Reset(); // Set True Add B latch
79:         i =
80:             (A_Reg -> Get().IsMinus()) +
81:             (B_Reg -> Get().IsMinus()) +
82:             (op_bin == OP_SUBTRACT);
83:         if(i & 1) {
84:             AComplement -> Set(); // Odd #: complement add
85:             CarryIn -> Set();
86:         }
87:         else {
88:             AComplement -> Reset(); // Even #: true add
89:             CarryIn -> Reset();
90:         }
91:
92:         adder_a = A_Reg -> Get();
93:
94:     } // End, units
95:     else {
96:         if(SubScanRing -> State() == SUB_SCAN_E) { // Extension latch?
97:             adder_a = BCD_0; // Yes, gate '0'
98:             // (if a compl then 9)
99:         }
100:        else {
101:            adder_a = A_Reg -> Get(); // No, use A data
102:        }
103:    }
104: } // End, SCAN 1
105:
106: else {
107:     assert(ScanRing -> State() == SCAN_3); // Else must be 3
108:     if(SubScanRing -> State() == SUB_SCAN_U) { // Units?
109:         b_temp = B_Reg -> Get(); // Yes. Invert sign
110:         b_temp = (b_temp & ~BIT_ZONE) |
111:             (b_temp.IsMinus() ? (BITA | BITB) : BITB);
112:         b_temp.SetOddParity();
113:         B_Reg -> Set(b_temp);
114:         AComplement -> Reset(); // Reset compl add
115:     }
116:     adder_a = BCD_0;
117: }
118:
119: // Run the right stuff thru the adder, and then thru the Assembly Channel
120:
121: Adder(addr_a,AComplement -> State(),
122:       B_Reg -> Get(),BComplement -> State());
123:
124: if(ZeroBalance -> State() &&
125:     (AdderResult & BIT_NUM) != (BCD_0 & ~BITC)) {
126:     ZeroBalance -> Reset();
127: }
128:
129: AssemblyChannel -> Select(
130:     AssemblyChannel -> AsmChannelWMB,
131:     AssemblyChannel -> AsmChannelZonesB,
132:     false,
```

```

133:         AssemblyChannel -> AsmChannelSignNone,
134:         AssemblyChannel -> AsmChannelNumAdder
135:     );
136:
137:     if(B_Reg -> Get().TestWM()) { // B Channel WM?
138:
139:         if(ScanRing -> State() == SCAN_3) { // 3rd scan latch?
140:             Store(AssemblyChannel -> Select()); // Store the last char
141:             Cycle(); // Modify B Addr -1
142:             IRingControl = true; // Done!
143:             return;
144:         }
145:
146:         if(AComplement -> State()) { // Complement add?
147:
148:             if(CarryOut -> State()) { // Carry too?
149:                 Store(AssemblyChannel -> Select()); // Store the last char
150:                 Cycle(); // Modify B addr -1
151:                 IRingControl = true; // Done! (no sign sw)
152:                 return;
153:             }
154:
155:             // What I *think* the following does is to reverse the B sign
156:             // where D_AR points, because SUB_SCAN_U causes the CPU to use
157:             // the DAR for the storage cycle, which starts a recomplement
158:             // cycle wherein the entire B field is recomplemented. This
159:             // can only happen if we are doing a complement add,
160:             // with no carry out.
161:
162:             SubScanRing -> Set(SUB_SCAN_U); // Set units latch
163:             ScanRing -> Set(SCAN_3); // Set 3rd scan latch
164:             BComplement -> Set(); // Set B compl add
165:             CarryIn -> Set(); // Set carry
166:             Store(AssemblyChannel -> Select()); // Store the last char
167:             Cycle(); // Finish the B cycle
168:             cycle_type = CYCLE_B; // Re-enter at B cycle
169:             return;
170:         }
171:
172:     else { // True add
173:         if(CarryOut -> State()) { // Set overflow on carry
174:             Overflow -> Set();
175:         }
176:         Store(AssemblyChannel -> Select()); // Store the last char
177:         Cycle(); // Modify B addr -1
178:         IRingControl = true; // Done!
179:         return;
180:     }
181: } // End B Ch WM
182:
183: else { // No B Ch WM
184:     CarryIn -> Set(CarryOut -> State()); // Set carry
185:     Store(AssemblyChannel -> Select()); // Store result char
186:     Cycle(); // Modify B address -1
187:     if(ScanRing -> State() == SCAN_1) { // Still 1st scan?
188:         if(A_Reg -> Get().TestWM()) { // A Ch WM?
189:             SubScanRing -> Set(SUB_SCAN_E); // Yes -- set extension
190:             cycle_type = CYCLE_B; // No more A cycles!
191:             return; // continue on
192:         }
193:         else { // No A CH WM
194:             SubScanRing -> Set(SUB_SCAN_B); // Set to body
195:             cycle_type = CYCLE_A; // A cycle next
196:             return;
197:         }
198:     }

```

```
199:         else {
200:             assert(ScanRing -> State() == SCAN_3);           // Must be!
201:             SubScanRing -> Set(SUB_SCAN_E);                 // Still in extension
202:             cycle_type = CYCLE_B;                           // Which means no A cyc
203:             return;
204:         }
205:     }
206: }
207:
208: void T1410CPU::InstructionZeroArith()
209: {
210:
211:     // Variable to maintain state between cycles
212:
213:     static int cycle_type;
214:     static int op_bin;
215:
216:     BCD a_temp,a_hold;
217:
218:     // If this is the first time in, set things up
219:
220:     if(LastInstructionReadout) {
221:         SubScanRing -> Set(SUB_SCAN_U);                  // Units
222:         ScanRing -> Set(SCAN_1);                      // Scan 1 (-1 mod)
223:
224:         AComplement -> Reset();
225:         BComplement -> Reset();
226:         CarryIn -> Reset();
227:         CarryOut -> Reset();
228:         ZeroBalance -> Set();                          // Assume 0 at start
229:
230:         cycle_type = CYCLE_A;                           // Set for 1st A cycle
231:         op_bin = Op_Reg -> Get().ToInt() & 0x3f;
232:         LastInstructionReadout = false;
233:     }
234:
235:     // Initial A cycle
236:
237:     if(cycle_type == CYCLE_A) {
238:         CycleRing -> Set(CYCLE_A);                   // Take an A Cycle
239:         *STAR = *A_AR;
240:         Readout();                                     // RO A field char.
241:         Cycle();                                       // Set AAR with mod
242:
243:         // Units or body latch Regen happens by itself in simulator
244:
245:         cycle_type = CYCLE_B;                         // Set for B cycle
246:         return;
247:     }
248:
249:     assert(cycle_type == CYCLE_B);                   // Has to be B cycle
250:
251:     CycleRing -> Set(CYCLE_B);                   // Take B Cycle
252:     if(SubScanRing -> State() == SUB_SCAN_U) {
253:         *STAR = *D_AR;                             // Read B units via DAR
254:     }
255:     else {
256:         *STAR = *B_AR;
257:     }
258:     Readout();                                     // Ro B Field Char
259:
260:     // If we are in the body/extension, we just insert 0's on the
261:     // A channel. Otherwise, we normalize the A field sign
262:
263:     a_temp = A_Reg -> Get();                      // Save to restore ...
264:     a_hold = a_temp;
```

```
265:     if(SubScanRing -> State() == SUB_SCAN_U) {
266:         if(op_bin == OP_ZERO_ADD) {                                // ZA - normalize sign
267:             a_temp = (a_temp & ~BIT_ZONE) |
268:                     (a_temp.IsMinus() ? BITB : (BITA | BITB));
269:         }
270:     } else {                                                 // ZS - invert sign
271:         a_temp = (a_temp & ~BIT_ZONE) |
272:                     (a_temp.IsMinus() ? (BITA | BITB) : BITB);
273:     }
274:     a_temp.SetOddParity();
275:
276:     // In the real machine, the sign fixing happens on the A Channel,
277:     // and not to the A Register itself. We'll cheat and use the A
278:     // register itself, so that the assembly channel can use the zones.
279:     // Then we will reset it later.
280:
281:     A_Reg -> Set(a_temp);
282: }
283:
284: if(SubScanRing -> State() == SUB_SCAN_E) {
285:     a_temp = BCD_0;
286: }
287:
288: // Run it thru the adder (kind of pointless in the simulator. Oh well.
289: Adder(a_temp, false, BCD_0, false);
290:
291: if(ZeroBalance -> State() &&
292:     (AdderResult & BIT_NUM) != (BCD_0 & ~BITC)) {
293:     ZeroBalance -> Reset();
294: }
295:
296: AssemblyChannel -> Select(
297:     AssemblyChannel -> AsmChannelWMB,
298:     (SubScanRing -> State() == SUB_SCAN_U ?
299:      AssemblyChannel -> AsmChannelZonesA :
300:      AssemblyChannel -> AsmChannelZonesNone),
301:     false,
302:     AssemblyChannel -> AsmChannelSignNone,           // Maybe signlatch someday
303:     AssemblyChannel -> AsmChannelNumAdder
304: );
305:
306: Store(AssemblyChannel -> Select());                         // Store the results
307: Cycle();                                                       // Finish the cycle
308:
309: A_Reg -> Set(a_hold);                                       // Reset a register
310:
311: if(B_Reg -> Get().TestWM()) {                                 // B Channel WM?
312:     IRingControl = true;                                      // Done!
313:     return;
314: }
315:
316: // Again, 1st Scan Latch and True Add latch regen all by themselves
317:
318: // CarryIn -> Set(CarryOut -> State());                   // Do carry as needed
319:
320: if(a_hold.TestWM()) {                                         // A Channel WM?
321:     SubScanRing -> Set(SUB_SCAN_E);                          // Yes -- extension
322:     cycle_type = CYCLE_B;                                     // B Cycle Next
323:     return;
324: }
325:
326: else {                                                       // No -- still body
327:     SubScanRing -> Set(SUB_SCAN_B);                          // A Cycle next
328:     cycle_type = CYCLE_A;
329:     return;
330: }
```

```

331:     }
332: }
333:
334: // Multiply instruction execution routine
335:
336: void T1410CPU::InstructionMultiply()
337: {
338:     static int cycle_type;
339:     static int cycle_subtype;
340:     static bool MultiplierDigitLatch;
341:
342:     BCD adder_a,b_temp;
343:
344:     if(LastInstructionReadout) {
345:         SubScanRing -> Set(SUB_SCAN_U);           // Set the Units latch
346:         ScanRing -> Set(SCAN_1);                // Set Scan to 1
347:         BComplement -> Reset();                 // True Add B
348:         AComplement -> Reset();                 // True Add A
349:         ZeroBalance -> Set();                  // Start as 0
350:         CarryIn -> Reset();                   // So single cycle
351:         CarryOut -> Reset();                  // looks good
352:         MultiplierDigitLatch = false;          // MDL latch reset
353:         SignLatch = false;                     // Start out as +
354:         cycle_type = CYCLE_A;                  // Start with A cycle
355:         cycle_subtype = 0;                      // No subtype
356:     }
357:
358:     if(cycle_type == CYCLE_A && cycle_subtype == 0) {
359:         CycleRing -> Set(CYCLE_A);           // Set for an A cycle
360:         if(SubScanRing -> State() == SUB_SCAN_U || 
361:             SubScanRing -> State() == SUB_SCAN_E) {
362:             *STAR = *C_AR;                    // A units, extension via CAR
363:         }
364:         else {
365:             assert(SubScanRing -> State() == SUB_SCAN_B); // A body via AAR
366:             *STAR = *A_AR;
367:         }
368:         Readout();
369:         Cycle();
370:         assert(ScanRing -> State() == SCAN_1); // Regen 1st Scan
371:         cycle_type = CYCLE_B;                  // Next cycle: B
372:         cycle_subtype = 0;
373:         return;
374:     } // End Cycle A
375:
376:     if(cycle_type == CYCLE_B && cycle_subtype == 0) {
377:
378:         CycleRing -> Set(CYCLE_B);           // Set for a B cycle
379:         if(SubScanRing -> State() == SUB_SCAN_U) {
380:             *STAR = *D_AR;                    // Read B units - DAR
381:         }
382:         else {
383:             *STAR = *B_AR;                  // Otherwise, use BAR
384:         }
385:         Readout();
386:
387:         if(SubScanRing -> State() == SUB_SCAN_U || 
388:             SubScanRing -> State() == SUB_SCAN_B || 
389:             SubScanRing -> State() == SUB_SCAN_E) {
390:
391:             Adder(BCD_0,AComplement->State(),      // Set up 0 in adder
392:                   BCD_0,BComplement->State());
393:             AssemblyChannel -> Select(            // No zones
394:                 AssemblyChannel -> AsmChannelWMB,
395:                 AssemblyChannel -> AsmChannelZonesNone,
396:                 false,

```

```

397:             AssemblyChannel -> AsmChannelSignNone,
398:             AssemblyChannel -> AsmChannelNumAdder);
399:             Store(AssemblyChannel -> Select()); // Store the 0
400:             Cycle(); // Modify B address by -1
401:             assert(!AComplement -> State()); // regen True add
402:
403:             if(SubScanRing -> State() == SUB_SCAN_E) { // Extension
404:                 CarryOut -> Reset(); // No carry
405:                 SubScanRing -> Set(SUB_SCAN_MQ); // Set MQ latch
406:                 assert(ScanRing -> State() == SCAN_1); // Regen 1st scan
407:                 cycle_type = CYCLE_B; // B Cycle next
408:                 cycle_subtype = 0;
409:                 return;
410:             } // Extension
411:
412:             assert(SubScanRing -> State() == SUB_SCAN_U || // Units or Body.
413:                   SubScanRing -> State() == SUB_SCAN_B);
414:
415:             if(A_Reg -> Get().TestWM()) { // A Channel WM?
416:                 SubScanRing -> Set(SUB_SCAN_E); // Set Extension latch
417:                 ScanRing -> Set(SCAN_N); // Set No Scan →
418:                 cycle_type = CYCLE_C; // C Cycle next
419:                 cycle_subtype = 0;
420:                 return;
421:             }
422:             else {
423:                 SubScanRing -> Set(SUB_SCAN_B); // Set Body latch
424:                 assert(ScanRing -> State() == SCAN_1); // Regen 1st scan
425:                 cycle_type = CYCLE_A; // A Cycle next
426:                 cycle_subtype = 0;
427:                 return;
428:             } // End: Units or body
429:
430:         } // End: Units, body or Extension
431:
432:     else {
433:         assert(SubScanRing -> State() == SUB_SCAN_MQ); // MQ
434:
435:         // As with add and subtract, the following is complicated on
436:         // the flow chart, but all it is really doing is checking to
437:         // see if the number of "-" values is even or odd
438:         if(((A_Reg -> Get().IsMinus()) + (B_Reg -> Get().IsMinus())) & 1) { // Odd: Set - sign
439:             CPU -> SignLatch = true;
440:         }
441:         else {
442:             CPU -> SignLatch = false;
443:         }
444:         → OK
445:         → DECODE → OK
446:         adder_a = BCD_9; // 9 on adder A ch.
447:         b_temp = B_Reg -> Get(); // Analyze B ch char
448:         b_temp = b_temp & BIT_NUM; // Look at just numerics
449:         if(b_temp == (BCD_0 & BIT_NUM)) { // Store 0 in B Field
450:             Store(AssemblyChannel -> Select (
451:                 AssemblyChannel -> AsmChannelWMB,
452:                 AssemblyChannel -> AsmChannelZonesNone,
453:                 false,
454:                 AssemblyChannel -> AsmChannelSignNone,
455:                 AssemblyChannel -> AsmChannelNumZero));
456:             Cycle();
457:             AComplement -> Reset(); // Set True add latch
458:             CarryOut -> Reset(); // Set No Carry
459:             if(B_Reg -> Get().TestWM()) { // Set 2nd Scan
460:                 ScanRing -> Set(SCAN_2);
461:             }
462:         }

```

(3)

```

463:             assert(ScanRing -> State() == SCAN_1); // Regen 1st scan
464:         }
465:     }
466:     else if(b_temp.ToInt() > 4) { // 5 - 9
467:         Store( AssemblyChannel -> Select ( // Store B numerics
468:             AssemblyChannel -> AsmChannelWMB,
469:             AssemblyChannel -> AsmChannelZonesNone,
470:             false,
471:             AssemblyChannel -> AsmChannelSignNone,
472:             AssemblyChannel -> AsmChannelNumB) );
473:         Cycle();
474:         AComplement -> Set(); // Set Complement
475:         Carryout -> Set(); // Set Carry
476:         ScanRing -> Set(SCAN_N); // Set No Scan
477:     }
478:
479:     else {
480:         Adder(adder_a,AComplement->State(),b_temp,BComplement->State());
481:         Store( AssemblyChannel -> Select ( // Store adder output
482:             AssemblyChannel -> AsmChannelWMB,
483:             AssemblyChannel -> AsmChannelZonesNone,
484:             false,
485:             AssemblyChannel -> AsmChannelSignNone,
486:             AssemblyChannel -> AsmChannelNumAdder) );
487:         Cycle();
488:         AComplement -> Reset(); // Set True Latch
489:         Carryout -> Reset(); // No Carry
490:         ScanRing -> Set(SCAN_N); // Set No Scan
491:     }
492:
493:     cycle_type = CYCLE_D;
494:     cycle_subtype = 0;
495:     return;
496:
497: } // MQ
498:
499: } // End B Cycle, subtype 0
500:
501: if(cycle_type == CYCLE_C) {
502:     CycleRing -> Set(CYCLE_C); // C Cycle
503:     *STAR = *C_AR; // Read out units pos
504:     Readout(); // Read storage
505:     Cycle(); // C is not modified
506:     ScanRing -> Set(SCAN_1); // B Cycle next time
507:     cycle_type = CYCLE_B; // Sub(Cycle D)
508:     cycle_subtype = 0;
509:     return;
510: } // End C Cycle
511:
512: if(cycle_type == CYCLE_D) {
513:     *STAR = *D_AR; // D Cycle
514:     Readout(); // 1st, 2nd scan
515:     if(ScanRing -> State() == SCAN_3) {
516:         Store( AssemblyChannel -> Select( // Store 0 & sign
517:             AssemblyChannel -> AsmChannelWMB,
518:             AssemblyChannel -> AsmChannelZonesNone,
519:             false,
520:             AssemblyChannel -> AsmChannelSignLatch,
521:             AssemblyChannel -> AsmChannelNumZero) );
522:     }
523:     // If memory Scan memory is regened (no store in emulator)
524:     // 3rd
525:     Cycle();
526:
527:     Carryout -> Set(AComplement -> State()); // Set carry state
528:

```

*Carry IN*

*Cycle modifying*

*Set(Cycle D)*

*Cycle D*

*reversed*

*Carry IN*

```

529:     if(ScanRing -> State() == SCAN_2) {           // Done if 2nd Scan
530:         IRingControl = true;
531:         return;
532:     }
533:
534:     if(ScanRing -> State() == SCAN_N || MultiplierDigitLatch) {
535:         SubScanRing -> Set(SUB_SCAN_U);           // Set Units latch
536:         ScanRing -> Set(SCAN_3);                 // Set 3rd scan
537:         BComplement -> Reset();                  // Set True Add B
538:         // AComplement regen'd
539:         cycle_type = CYCLE_A;                    // A Cycle next
540:         cycle_subtype = 1;                      // 2nd kind of A cycle
541:         return;
542:     }
543:
544:     // 1st or 3rd scan, MDL latch reset...
545:
546:     SubScanRing -> Set(SUB_SCAN_MQ);           // Set MQ latch
547:     ScanRing -> Set(SCAN_3);                  // Set 3rd scan
548:     BComplement -> Reset();                  // True add B latch
549:     // Acomplement regen'd
550:     cycle_type = CYCLE_B;                    // B Cycle next
551:     cycle_subtype = 1;                      // 2nd kind of B cycle
552:     return;
553: } // End D Cycle
554:
555: if(cycle_type == CYCLE_A && cycle_subtype == 1) {   Cycle B → Set(Cycle-A)
556:     if(SubScanRing -> State() == SUB_SCAN_U ||      // Units or Extension?
557:         SubScanRing -> State() == SUB_SCAN_E) {       // Yes, use CAR for address
558:         *STAR = *C_AR;
559:     }
560:     else {
561:         *STAR = *A_AR;
562:     }
563:     Readout();                                     // Read out A field
564:     Cycle();
565:     assert(ScanRing -> State() == SCAN_3);        // Regen 3rd scan
566:     BComplement -> Reset();                      // True add B latch
567:     // A Complement regen'd
568:     cycle_type = CYCLE_B;                        // B Cycle next
569:     cycle_subtype = 1;                          // 2nd kind of B cycle
570:     return;
571: }
572:
573: if(cycle_type == CYCLE_B && cycle_subtype == 1) {   Cycle B → Set(Cycle-B)
574:
575:     if(SubScanRing -> State() == SUB_SCAN_U) {      *STAR = *D_AR;
576:         *STAR = *D_AR; 
577:     }
578:     else {
579:         *STAR = *B_AR;
580:     }
581:     Readout();
582:
583:     if(SubScanRing -> State() == SUB_SCAN_MQ) {
584:         if(AComplement -> State()) {
585:             adder_a = BCD_0;                         // Insert 0 on A
586:             // Carry is also set
587:             // Will add 1 to B
588:         }
589:         else {
590:             adder_a = BCD_9;                         // Insert 9 on A
591:             // Will decrement B
592:         }
593:         b_temp = B_Reg -> Get();                // Get B data
594:         b_temp = b_temp & BIT_NUM;               // Numeric bits

```

```

595:         if(b_temp == (BCD_0 & BIT_NUM) && !(AComplement -> State()) ) {
596:             Store( AssemblyChannel -> Select'()           // Store 0, no zones
597:                   AssemblyChannel -> AsmChannelWMB,
598:                   AssemblyChannel -> AsmChannelZonesNone,
599:                   false,
600:                   AssemblyChannel -> AsmChannelSignNone,
601:                   AssemblyChannel -> AsmChannelNumZero) );
602:             Cycle();
603:             assert(! (AComplement -> State()));           // True Add (was already)
604:             Carryout -> Reset();                         // Reset Carry
605:             if(B_Reg -> Get().TestWM()) {                  // B WM?
606:                 IRingControl = true;                      // Yes. All done.
607:                 return;
608:             }
609:             assert(ScanRing -> State() == SCAN_3);        // Regen 3rd Scan latch
610:             assert!(BComplement -> State());              // Regen B True add
611:             cycle_type = CYCLE_D;                          // Next cycle D
612:             cycle_subtype = 0;
613:             return;
614:         }
615:     }
616:
617:     if(AComplement -> State()) &&
618:         (b_temp == (BCD_0 & BIT_NUM) || b_temp.ToInt() < 5) {
619:         Store( AssemblyChannel -> Select'()           // Store B (no zones)
620:               AssemblyChannel -> AsmChannelWMB,
621:               AssemblyChannel -> AsmChannelZonesNone,
622:               false,
623:               AssemblyChannel -> AsmChannelSignNone,
624:               AssemblyChannel -> AsmChannelNumB) );
625:         Cycle();
626:         AComplement -> Reset();                        // Set True Add latch
627:         Carryout -> Reset();
628:         SubScanRing -> Set(SUB_SCAN_U);                // Set Units latch
629:         assert(ScanRing -> State() == SCAN_3);          // Regen 3rd scan
630:         BComplement -> Reset();                        // Set True Add B latch
631:         cycle_type = CYCLE_A;                          // Next Cycle: A
632:         cycle_subtype = 1;                            // second kind of A
633:         return;
634:     }
635:     assert(b_temp != (BCD_0 & BIT_NUM) && b_temp.ToInt() >= 5);
636:
637:     if(b_temp.ToInt() < 9 && AComplement -> State()) {
638:         Adder(addr_a,AComplement -> State(),
639:               B_Reg -> Get(),BComplement -> State());
640:         Store( AssemblyChannel -> Select'()           // Store adder, no zones
641:               AssemblyChannel -> AsmChannelWMB,
642:               AssemblyChannel -> AsmChannelZonesNone,
643:               false,
644:               AssemblyChannel -> AsmChannelSignNone,
645:               AssemblyChannel -> AsmChannelNumAdder) );
646:         Cycle();
647:         AComplement -> Set();                          // Set Complement latch
648:         Carryout -> Set();                          // Set carry latch
649:         SubScanRing -> Set(SUB_SCAN_U);                // Set Units latch
650:         assert(ScanRing -> State() == SCAN_3);          // Regen 3rd scan
651:         BComplement -> Reset();                        // True add B
652:         cycle_type = CYCLE_A;                          // Next Cycle A
653:         cycle_subtype = 1;                            // second kind of A
654:         return;
655:     }
656:
657:     if(!(AComplement -> State())) {                  // True latch on?
658:         Store( AssemblyChannel -> Select'()           // Store B, no zones
659:               AssemblyChannel -> AsmChannelWMB,
660:               
```

```

661:             AssemblyChannel -> AsmChannelZonesNone,
662:             false,
663:             AssemblyChannel -> AsmChannelSignNone,
664:             AssemblyChannel -> AsmChannelNumB) );
665:             Cycle();
666:             AComplement -> Set(); // Set Complement latch
667:             CarryOut -> Set(); // Set carry latch
668:             SubScanRing -> Set(SUB_SCAN_U); // Set Units latch
669:             assert(ScanRing -> State() == SCAN_3); // Regen 3rd scan
670:             BComplement -> Reset(); // True add B
671:             cycle_type = CYCLE_A; // Next Cycle A
672:             cycle_subtype = 1; // second kind of A
673:             return;
674:         }
675:
676:         assert(b_temp.ToInt() == 9 && AComplement -> State()); ;
677:
678:         Store(AssemblyChannel -> Select ( // Store 0 or B, no zones
679:             AssemblyChannel -> AsmChannelWMB,
680:             AssemblyChannel -> AsmChannelZonesNone,
681:             false,
682:             AssemblyChannel -> AsmChannelSignNone,
683:             (ZeroBalance -> State()) ?
684:                 AssemblyChannel -> AsmChannelNumZero :
685:                 AssemblyChannel -> AsmChannelNumB) );
686:             Cycle();
687:             if(B_Reg -> Get().TestWM()) { // B WM?
688:                 AComplement -> Reset(); // Set True Add latch
689:                 MultiplierDigitLatch = true; // Set MDL latch
690:             }
691:             else {
692:                 AComplement -> Set(); // No. Set Complement
693:             }
694:             assert(ScanRing -> State() == SCAN_3); // Regen 3rd scan
695:             BComplement -> Reset(); // True Add B
696:             cycle_type = CYCLE_D; // D Cycle Next
697:             cycle_subtype = 0;
698:             return;
699:         } // End: MQ
700:
701:         if(SubScanRing -> State() == SUB_SCAN_E) {
702:             if(AComplement -> State()) {
703:                 adder_a = BCD_9; // Insert 9 on A
704:             }
705:             else {
706:                 adder_a = BCD_0; // Insert 0 on A
707:             }
708:         }
709:         else {
710:             adder_a = A_Reg -> Get(); // Gate A Ch to adder
711:         }
712:
713:         Adder(addr_a, AComplement -> State(),
714:               B_Reg -> Get(), BComplement -> State());
715:
716:         Store ( AssemblyChannel -> Select ( // Adder, B Zones to Asm
717:             AssemblyChannel -> AsmChannelWMB,
718:             AssemblyChannel -> AsmChannelZonesB,
719:             false,
720:             AssemblyChannel -> AsmChannelSignNone,
721:             AssemblyChannel -> AsmChannelNumAdder) );
722:             Cycle();
723:
724:             if((AdderResult & BIT_NUM) == (BCD_0 & ~BITC)) { // Check Zero Balance
725:                 ZeroBalance -> Reset();
726:             }

```

(2)

```

727: if(SubScanRing -> State() == SUB_SCAN_E) { // Extension?
728:     CarryOut -> Set(AComplement -> State()); // Yes. Possibly set carry
729:     if(MultiplierDigitLatch) { // MDL ?
730:         IRingControl = true; // Yes - Done
731:         return;
732:     }
733:     SubScanRing -> Set(SUB_SCAN_MQ); // No - set MQ latch
734:     assert(ScanRing -> State() == SCAN_3); // Regen 3rd scan
735:     BComplement -> Reset(); // True Add B
736:     cycle_type = CYCLE_B; // Take another B cycle
737:     cycle_subtype = 1; // also subtype 1
738:     return;
739: }
740:
741:
742: assert(SubScanRing -> State() == SUB_SCAN_U || // Must be Units or Body
743:        SubScanRing -> State() == SUB_SCAN_B); // Adder has already set CarryOut appropriately
744: CARRYIN -> SET(CARRYOUT -> STATE);
745: // Adder has already set CarryOut appropriately
746:
747: if(A_Reg -> Get().TestWM()) { Setc } // A Channel WM ?
748:     SubScanRing -> State() == SUB_SCAN_E; // Yes. Set Extension
749:     assert(ScanRing -> State() == SCAN_3); // Regen 3rd scan
750:     cycle_type = CYCLE_B; // Take another B Cycle
751:     cycle_subtype = 1; // Also subtype 1
752:     return;
753:
754: else { Set( ) } // No WM. Body.
755:     SubScanRing -> State() == SUB_SCAN_B; // Regen 3rd scan
756:     assert(ScanRing -> State() == SCAN_3); // True Add B
757:     BComplement -> Reset(); // Take an A cycle next
758:     cycle_type = CYCLE_A; // Subtype 1
759:     cycle_subtype = 1;
760:     return;
761:
762: } // End: B Cycle, subtype 1
763:
764: }
765:
766: do
767:
768:
769:
770:

```



```
67:     // Note that we don't do the address modification cycle yet, as we
68:     // have to store the result back where BAR points!
69:
70:     // The following looks complicated on the flow chart in the
71:     // IBM CE Instructional materials 1411 Processing Unit Instructions
72:     // on page 11. But, the end result is that if you have an EVEN number
73:     // of: -A, -B and SUBTRACT you do a TRUE ADD cycle, and ODD number and
74:     // you do a COMPLEMENT ADD cycle
75:
76:     if(ScanRing -> State() == SCAN_1) {           // Is 1st scan ring on?
77:         if(SubScanRing -> State() == SUB_SCAN_U) { // Units latch?
78:             BComplement -> Reset();                // Set True Add B latch
79:             i =
80:                 (A_Reg -> Get().IsMinus()) +
81:                 (B_Reg -> Get().IsMinus()) +
82:                 (op_bin == OP_SUBTRACT);
83:             if(i & 1) {
84:                 AComplement -> Set();                  // Odd #: complement add
85:                 CarryIn -> Set();
86:             }
87:             else {
88:                 AComplement -> Reset();                // Even #: true add
89:                 CarryIn -> Reset();
90:             }
91:
92:             adder_a = A_Reg -> Get();
93:
94:         } // End, units
95:     else {
96:         if(SubScanRing -> State() == SUB_SCAN_E) { // Extension latch?
97:             adder_a = BCD_0;                         // Yes, gate '0'
98:                                         // (if a compl then 9)
99:
100:        }
101:        else {
102:            adder_a = A_Reg -> Get();              // No, use A data
103:        }
104:    } // End, SCAN 1
105:
106: else {
107:     assert(ScanRing -> State() == SCAN_3);          // Else must be 3
108:     if(SubScanRing -> State() == SUB_SCAN_U) {        // Units?
109:         b_temp = B_Reg -> Get();                      // Yes. Invert sign
110:         b_temp = (b_temp & ~BIT_ZONE) |
111:             (b_temp.IsMinus() ? (BITA | BITB) : BITB);
112:         b_temp.SetOddParity();
113:         B_Reg -> Set(b_temp);
114:         AComplement -> Reset();                      // Reset compl add
115:     }
116:     adder_a = BCD_0;
117: }
118:
119: // Run the right stuff thru the adder, and then thru the Assembly Channel
120:
121: Adder(adder_a,AComplement -> State(),
122:       B_Reg -> Get(),BComplement -> State());
123:
124: if(ZeroBalance -> State() &&
125:     (AdderResult & BIT_NUM) != (BCD_0 & ~BITC)) {
126:     ZeroBalance -> Reset();
127: }
128:
129: AssemblyChannel -> Select(
130:     AssemblyChannel -> AsmChannelWMB,
131:     AssemblyChannel -> AsmChannelZonesB,
132:     false,
```

```

133:         AssemblyChannel -> AsmChannelSignNone,
134:         AssemblyChannel -> AsmChannelNumAdder
135:     );
136:
137:     if(B_Reg -> Get().TestWM()) { // B Channel WM?
138:
139:         if(ScanRing -> State() == SCAN_3) { // 3rd scan latch?
140:             Store(AssemblyChannel -> Select()); // Store the last char
141:             Cycle(); // Modify B Addr -1
142:             IRingControl = true; // Done!
143:             return;
144:         }
145:
146:         if(AComplement -> State()) { // Complement add?
147:
148:             if(CarryOut -> State()) { // Carry too?
149:                 Store(AssemblyChannel -> Select()); // Store the last char
150:                 Cycle(); // Modify B addr -1
151:                 IRingControl = true; // Done! (no sign sw)
152:                 return;
153:             }
154:
155:             // What I *think* the following does is to reverse the B sign
156:             // where D_AR points, because SUB_SCAN_U causes the CPU to use
157:             // the DAR for the storage cycle, which starts a recomplement
158:             // cycle wherein the entire B field is recomplemented. This
159:             // can only happen if we are doing a complement add,
160:             // with no carry out.
161:
162:             SubScanRing -> Set(SUB_SCAN_U); // Set units latch
163:             ScanRing -> Set(SCAN_3); // Set 3rd scan latch
164:             BComplement -> Set(); // Set B compl add
165:             CarryIn -> Set(); // Set carry
166:             Store(AssemblyChannel -> Select()); // Store the last char
167:             Cycle(); // Finish the B cycle
168:             cycle_type = CYCLE_B; // Re-enter at B cycle
169:             return;
170:         }
171:
172:     else { // True add
173:         if(CarryOut -> State()) { // Set overflow on carry
174:             Overflow -> Set();
175:         }
176:         Store(AssemblyChannel -> Select()); // Store the last char
177:         Cycle(); // Modify B addr -1
178:         IRingControl = true; // Done!
179:         return;
180:     }
181: } // End B Ch WM
182:
183: else { // No B Ch WM
184:     CarryIn -> Set(CarryOut -> State()); // Set carry
185:     Store(AssemblyChannel -> Select()); // Store result char
186:     Cycle(); // Modify B address -1
187:     if(ScanRing -> State() == SCAN_1) { // Still 1st scan?
188:         if(A_Reg -> Get().TestWM()) { // A Ch WM?
189:             SubScanRing -> Set(SUB_SCAN_E); // Yes -- set extension
190:             cycle_type = CYCLE_B; // No more A cycles!
191:             return; // continue on
192:         }
193:         else { // No A CH WM
194:             SubScanRing -> Set(SUB_SCAN_B); // Set to body
195:             cycle_type = CYCLE_A; // A cycle next
196:             return;
197:         }
198:     }

```

```
199:         else {
200:             assert(ScanRing -> State() == SCAN_3);           // Must be!
201:             SubScanRing -> Set(SUB_SCAN_E);                 // Still in extension
202:             cycle_type = CYCLE_B;                           // Which means no A cyc
203:             return;
204:         }
205:     }
206: }
207:
208: void T1410CPU::InstructionZeroArith()
209: {
210:
211:     // Variable to maintain state between cycles
212:
213:     static int cycle_type;
214:     static int op_bin;
215:
216:     BCD a_temp,a_hold;
217:
218:     // If this is the first time in, set things up
219:
220:     if(LastInstructionReadout) {
221:         SubScanRing -> Set(SUB_SCAN_U);                  // Units
222:         ScanRing -> Set(SCAN_1);                      // Scan 1 (-1 mod)
223:
224:         AComplement -> Reset();
225:         BComplement -> Reset();
226:         CarryIn -> Reset();
227:         CarryOut -> Reset();
228:         ZeroBalance -> Set();                          // Assume 0 at start
229:
230:         cycle_type = CYCLE_A;                           // Set for 1st A cycle
231:         op_bin = Op_Reg -> Get().ToInt() & 0x3f;
232:         LastInstructionReadout = false;
233:     }
234:
235:     // Initial A cycle
236:
237:     if(cycle_type == CYCLE_A) {
238:         CycleRing -> Set(CYCLE_A);                   // Take an A Cycle
239:         *STAR = *A_AR;
240:         Readout();                                     // RO A field char.
241:         Cycle();                                       // Set AAR with mod
242:
243:         // Units or body latch Regen happens by itself in simulator
244:
245:         cycle_type = CYCLE_B;                         // Set for B cycle
246:         return;
247:     }
248:
249:     assert(cycle_type == CYCLE_B);                   // Has to be B cycle
250:
251:     CycleRing -> Set(CYCLE_B);                   // Take B Cycle
252:     if(SubScanRing -> State() == SUB_SCAN_U) {      // Read B units via DAR
253:         *STAR = *D_AR;
254:     }
255:     else {
256:         *STAR = *B_AR;
257:     }
258:     Readout();                                     // Ro B Field Char
259:
260:     // If we are in the body/extension, we just insert 0's on the
261:     // A channel. Otherwise, we normalize the A field sign
262:
263:     a_temp = A_Reg -> Get();                      // Save to restore ...
264:     a_hold = a_temp;
```

```
265:     if(SubScanRing -> State() == SUB_SCAN_U) {
266:         if(op_bin == OP_ZERO_ADD) {                                // ZA - normalize sign
267:             a_temp = (a_temp & ~BIT_ZONE) |
268:                     (a_temp.IsMinus() ? BITB : (BITA | BITB));
269:         }
270:     } else {                                                 // ZS - invert sign
271:         a_temp = (a_temp & ~BIT_ZONE) |
272:                     (a_temp.IsMinus() ? (BITA | BITB) : BITB);
273:     }
274:     a_temp.SetOddParity();
275:
276:     // In the real machine, the sign fixing happens on the A Channel,
277:     // and not to the A Register itself. We'll cheat and use the A
278:     // register itself, so that the assembly channel can use the zones.
279:     // Then we will reset it later.
280:
281:     A_Reg -> Set(a_temp);
282: }
283:
284: if(SubScanRing -> State() == SUB_SCAN_E) {
285:     a_temp = BCD_0;
286: }
287:
288: // Run it thru the adder (kind of pointless in the simulator. Oh well.
289: Adder(a_temp, false, BCD_0, false);
290:
291: if(ZeroBalance -> State() &&
292:     (AdderResult & BIT_NUM) != (BCD_0 & ~BITC)) {
293:     ZeroBalance -> Reset();
294: }
295:
296: AssemblyChannel -> Select(
297:     AssemblyChannel -> AsmChannelWMB,
298:     (SubScanRing -> State() == SUB_SCAN_U ?
299:      AssemblyChannel -> AsmChannelZonesA :
300:      AssemblyChannel -> AsmChannelZonesNone),
301:     false,
302:     AssemblyChannel -> AsmChannelSignNone,           // Maybe signlatch someday
303:     AssemblyChannel -> AsmChannelNumAdder
304: );
305:
306: Store(AssemblyChannel -> Select());                         // Store the results
307: Cycle();                                                       // Finish the cycle
308:
309: A_Reg -> Set(a_hold);                                       // Reset a register
310:
311: if(B_Reg -> Get().TestWM()) {                                 // B Channel WM?
312:     IRingControl = true;                                      // Done!
313:     return;
314: }
315:
316: }
317:
318: // Again, 1st Scan Latch and True Add latch regen all by themselves
319:
320: CarryIn -> Set(CarryOut -> State());                      // Do carry as needed
321:
322: if(a_hold.TestWM()) {                                         // A Channel WM?
323:     SubScanRing -> Set(SUB_SCAN_E);                          // Yes -- extension
324:     cycle_type = CYCLE_B;                                     // B Cycle Next
325:     return;
326: }
327: else {                                                       // No -- still body
328:     SubScanRing -> Set(SUB_SCAN_B);                          // A Cycle next
329:     cycle_type = CYCLE_A;
330:     return;
```

```

331:     }
332: }
333:
334: // Multiply instruction execution routine
335:
336: void T1410CPU::InstructionMultiply()
337: {
338:     static int cycle_type;
339:     static int cycle_subtype;
340:
341:     BCD adder_a,b_temp;
342:
343:     if(LastInstructionReadout) {
344:         SubScanRing -> Set(SUB_SCAN_U);
345:         ScanRing -> Set(SCAN_1);
346:         BComplement -> Reset();
347:         AComplement -> Reset();
348:         cycle_type = CYCLE_A;
349:         cycle_subtype = 0;
350:     }
351:
352:     if(cycle_type == CYCLE_A) {
353:         CycleRing -> Set(CYCLE_A); // Set for an A cycle
354:         if(SubScanRing -> State() == SUB_SCAN_U ||
355:             SubScanRing -> State() == SUB_SCAN_E) {
356:             *STAR = *C_AR; // A units, extension via CAR
357:         }
358:         else {
359:             assert(SubScanRing -> State() == SUB_SCAN_B);
360:             *STAR = *A_AR; // A body via AAR
361:         }
362:         assert(ScanRing -> State() == SCAN_1); // Regen 1st Scan
363:         cycle_type = CYCLE_B; // Next cycle: B
364:         cycle_subtype = 0;
365:         return;
366:     } // Cycle A
367:
368:     if(cycle_type == CYCLE_B && cycle_subtype == 0) {
369:
370:         CycleRing -> Set(CYCLE_B); // Set for a B cycle
371:         if(SubScanRing -> State() == SUB_SCAN_U) {
372:             *STAR = *D_AR; // Read B units - DAR
373:         }
374:         else {
375:             *STAR = *B_AR; // Otherwise, use BAR
376:         }
377:
378:         if(SubScanRing -> State() == SUB_SCAN_U ||
379:             SubScanRing -> State() == SUB_SCAN_B ||
380:             SubScanRing -> State() == SUB_SCAN_E) {
381:
382:             Adder(BCD_0,AComplement->State(),
383:                   BCD_0,BComplement->State());
384:             AssemblyChannel -> Select( // Store the 0
385:                 AssemblyChannel -> AsmChannelWMB,
386:                 AssemblyChannel -> AsmChannelZonesNone,
387:                 false,
388:                 AssemblyChannel -> AsmChannelSignNone,
389:                 AssemblyChannel -> AsmChannelNumAdder);
390:             Store(AssemblyChannel -> Select()); // Modify B address by -1
391:             Cycle(); // regen True add
392:             assert(!AComplement -> State()); // regen True add
393:
394:             if(SubScanRing -> State() == SUB_SCAN_E) { // Extension
395:                 CarryOut -> Reset(); // No carry
396:                 SubScanRing -> Set(SUB_SCAN_MQ); // Set MQ latch

```

Handwritten annotations:

- Line 343: A checkmark is present on the left.
- Line 344: A handwritten note "Set(0)" is above the code, with arrows pointing to "Set" and "0".
- Line 345: A handwritten note "Set(1)" is above the code, with arrows pointing to "Set" and "1".
- Line 346: A handwritten note "Reset()" is above the code, with arrows pointing to "Reset".
- Line 347: A handwritten note "Reset()" is above the code, with arrows pointing to "Reset".
- Line 348: A handwritten note "cycle\_type = CYCLE\_A;" is above the code, with arrows pointing to "cycle\_type" and "CYCLE\_A".
- Line 349: A handwritten note "cycle\_subtype = 0;" is above the code, with arrows pointing to "cycle\_subtype" and "0".
- Line 352: A checkmark is present on the left.
- Line 353: A handwritten note "Set(CYCLE\_A)" is above the code, with arrows pointing to "Set" and "CYCLE\_A".
- Line 354: A handwritten note "assert(SubScanRing -> State() == SUB\_SCAN\_U || SubScanRing -> State() == SUB\_SCAN\_E)" is above the code, with arrows pointing to "assert", "SubScanRing", "State()", "SUB\_SCAN\_U", and "SUB\_SCAN\_E".
- Line 356: A handwritten note "\*STAR = \*C\_AR;" is above the code, with arrows pointing to "\*STAR" and "\*C\_AR".
- Line 358: A checkmark is present on the left.
- Line 359: A handwritten note "assert(SubScanRing -> State() == SUB\_SCAN\_B)" is above the code, with arrows pointing to "assert", "SubScanRing", "State()", and "SUB\_SCAN\_B".
- Line 360: A handwritten note "\*STAR = \*A\_AR;" is above the code, with arrows pointing to "\*STAR" and "\*A\_AR".
- Line 362: A handwritten note "assert(ScanRing -> State() == SCAN\_1)" is above the code, with arrows pointing to "assert", "ScanRing", "State()", and "SCAN\_1".
- Line 363: A handwritten note "cycle\_type = CYCLE\_B;" is above the code, with arrows pointing to "cycle\_type" and "CYCLE\_B".
- Line 364: A handwritten note "cycle\_subtype = 0;" is above the code, with arrows pointing to "cycle\_subtype" and "0".
- Line 366: A checkmark is present on the left.
- Line 368: A checkmark is present on the left.
- Line 370: A handwritten note "Set(CYCLE\_B)" is above the code, with arrows pointing to "Set" and "CYCLE\_B".
- Line 371: A handwritten note "assert(SubScanRing -> State() == SUB\_SCAN\_U)" is above the code, with arrows pointing to "assert", "SubScanRing", "State()", and "SUB\_SCAN\_U".
- Line 372: A handwritten note "\*STAR = \*D\_AR;" is above the code, with arrows pointing to "\*STAR" and "\*D\_AR".
- Line 374: A checkmark is present on the left.
- Line 375: A handwritten note "\*STAR = \*B\_AR;" is above the code, with arrows pointing to "\*STAR" and "\*B\_AR".
- Line 378: A checkmark is present on the left.
- Line 379: A handwritten note "assert(SubScanRing -> State() == SUB\_SCAN\_U || SubScanRing -> State() == SUB\_SCAN\_B || SubScanRing -> State() == SUB\_SCAN\_E)" is above the code, with arrows pointing to "assert", "SubScanRing", "State()", "SUB\_SCAN\_U", "||", "SubScanRing", "State()", "SUB\_SCAN\_B", "||", "SubScanRing", "State()", "SUB\_SCAN\_E".
- Line 382: A handwritten note "Adder(BCD\_0,AComplement->State(), BCD\_0,BComplement->State());" is above the code, with arrows pointing to "Adder", "BCD\_0", "AComplement", "BCD\_0", "BComplement".
- Line 384: A handwritten note "AssemblyChannel -> Select(" is above the code, with arrows pointing to "AssemblyChannel", "Select".
- Line 385: A handwritten note "AssemblyChannel -> AsmChannelWMB," is above the code, with arrows pointing to "AssemblyChannel", "AsmChannelWMB".
- Line 386: A handwritten note "AssemblyChannel -> AsmChannelZonesNone," is above the code, with arrows pointing to "AssemblyChannel", "AsmChannelZonesNone".
- Line 387: A handwritten note "false," is above the code, with arrows pointing to "false".
- Line 388: A handwritten note "AssemblyChannel -> AsmChannelSignNone," is above the code, with arrows pointing to "AssemblyChannel", "AsmChannelSignNone".
- Line 389: A handwritten note "AssemblyChannel -> AsmChannelNumAdder);" is above the code, with arrows pointing to "AssemblyChannel", "AsmChannelNumAdder".
- Line 390: A handwritten note "Store(AssemblyChannel -> Select());" is above the code, with arrows pointing to "Store", "AssemblyChannel", "Select".
- Line 391: A handwritten note "// Modify B address by -1" is above the code, with arrows pointing to the comment.
- Line 392: A handwritten note "Cycle();" is above the code, with arrows pointing to "Cycle".
- Line 393: A handwritten note "assert(!AComplement -> State());" is above the code, with arrows pointing to "assert", "AComplement", "State".
- Line 394: A handwritten note "if(SubScanRing -> State() == SUB\_SCAN\_E) { // Extension" is above the code, with arrows pointing to "if", "SubScanRing", "State()", "SUB\_SCAN\_E", and the comment.
- Line 395: A handwritten note "CarryOut -> Reset(); // No carry" is above the code, with arrows pointing to "CarryOut", "Reset", and the comment.
- Line 396: A handwritten note "SubScanRing -> Set(SUB\_SCAN\_MQ); // Set MQ latch" is above the code, with arrows pointing to "SubScanRing", "Set", "SUB\_SCAN\_MQ", and the comment.

```

397:         assert(ScanRing -> State() == SCAN_1); // Regen 1st scan
398:         cycle_type = CYCLE_B; // B Cycle next
399:         cycle_subtype = 0;
400:         return;
401:     } // Extension
402:
403:     assert(SubScanRing -> State() == SUB_SCAN_U || // Units or Body
404:            SubScanRing -> State() == SUB_SCAN_B); // Units or Body
405:
406:     if(A_Reg -> Get().TestWM()) { // A Channel WM?
407:         SubScanRing -> Set(SUB_SCAN_E); // Set Extension latch
408:         ScanRing -> Set(SCAN_N); // Set No Scan
409:         cycle_type = CYCLE_C; // C Cycle next
410:         cycle_subtype = 0;
411:         return;
412:     }
413:     else {
414:         SubScanRing -> Set(SUB_SCAN_B); // Set Body latch
415:         assert(ScanRing -> State() == SCAN_1); // Regen 1st scan
416:         cycle_type = CYCLE_A; // A Cycle next
417:         cycle_subtype = 0;
418:         return;
419:     } // End: Units or body
420:
421: } // End: Units, body or Extension
422:
423: else {
424:     assert(SubScanRing -> State() == SUB_SCAN_MQ); // MQ
425:
426:     // As with add and subtract, the following is complicated on
427:     // the flow chart, but all it is really doing is checking to
428:     // see if the number of "-" values is even or odd
429:
430:     if(((A_Reg -> Get().IsMinus()) + (B_Reg -> Get().IsMinus())) & 1) {
431:         CPU -> SignLatch = true; // Odd: Set - sign
432:     }
433:     else {
434:         CPU -> SignLatch = false;
435:     }
436:
437:     adder_a = BCD_9; // 9 on adder A ch.
438:     b_temp = B_Reg -> Get(); // Analyze B ch char
439:     b_temp = b_temp & BIT_NUM; // Look at just numerics
440:     if(b_temp == (BCD_0 & BIT_NUM)) { // Store 0 in B Field
441:         Store(AssemblyChannel -> Select (
442:             AssemblyChannel -> AsmChannelWMB,
443:             AssemblyChannel -> AsmChannelZonesNone,
444:             false,
445:             AssemblyChannel -> AsmChannelSignNone,
446:             AssemblyChannel -> AsmChannelNumZero));
447:         Cycle();
448:         AComplement -> Reset(); // Set True add latch
449:         CarryOut -> Reset(); // Set No Carry
450:         if(B_Reg -> Get().TestWM()) {
451:             ScanRing -> Set(SCAN_2); // Set 2nd Scan
452:         }
453:         else {
454:             assert(ScanRing -> State() == SCAN_1); // Regen 1st scan
455:         }
456:     }
457:     else if(b_temp.ToInt() & BIT4) { // 5 - 9
458:         Store(AssemblyChannel -> Select ( // Store B numerics
459:             AssemblyChannel -> AsmChannelWMB,
460:             AssemblyChannel -> AsmChannelZonesNone,
461:             false,
462:             AssemblyChannel -> AsmChannelSignNone,

```

```
463:             AssemblyChannel -> AsmChannelNumB) );
464:             Cycle();
465:             AComplement -> Set();                                // Set Complement
466:             CarryOut -> Set();                                 // Set Carry
467:             ScanRing -> Set(SCAN_N);                           // Set No Scan
468:         }
469:
470:     else {
471:         Adder(adder_a,AComplement->State(),b_temp,BComplement->State());
472:         Store( AssemblyChannel -> Select (           // Store adder output
473:             AssemblyChannel -> AsmChannelWMB,
474:             AssemblyChannel -> AsmChannelZonesNone,
475:             false,
476:             AssemblyChannel -> AsmChannelSignNone,
477:             AssemblyChannel -> AsmChannelNumAdder) );
478:         Cycle();
479:         AComplement -> Reset();                               // Set True Latch
480:         CarryOut -> Reset();                                // No Carry
481:         ScanRing -> Set(SCAN_N);                           // Set No Scan
482:     }
483:
484:     cycle_type = CYCLE_D;
485:     cycle_subtype = 0;
486:     return;
487:
488: } // MQ
489:
490: // B Cycle, subtype 0
491:
492: if(cycle_type == CYCLE_C) {
493:     CycleRing -> Set(CYCLE_C);                          // C Cycle
494:     *STAR = *C_AR;                                       // Read out units pos
495:     Readout();                                         // Read storage
496:     Cycle();                                            // C is not modified
497:     ScanRing -> Set(SCAN_1);                           // B Cycle next time
498:     cycle_type = CYCLE_B;
499:     cycle_subtype = 0;
500:     return;
501: }
502:
503: }
504:
505:
```

```
1: //-----
2: #ifndef UI1410CPUTH
3: #define UI1410CPUTH
4:
5: #include "ubcd.h"
6:
7: extern long ten_thousands[],thousands[],hundreds[],tens[];
8: extern long scan_mod[];
9: extern unsigned char sign_normalize_table[];
10: extern unsigned char sign_complement_table[];
11: extern unsigned char sign_negative_table[];
12:
13: struct OpCodeCommonLines {
14:     unsigned short ReadOut;
15:     unsigned short Operational;
16:     unsigned short Control;
17: };
18:
19:
20: extern struct OpCodeCommonLines OpCodeTable[64];
21:
22: extern int IndexRegisterLookup [];
23:
24: //
25: // Classes (types) used to implement the emulator, including the
26: // final class defining what is in the CPU, T1410CPU.
27:
28: //
29: // Abstract class designed to build lists of objects affected by Program
30: // Reset and Computer Reset
31: //
32:
33: class TCpuObject : public TObject {
34:
35: public:
36:     TCpuObject();                                // Constructor to init data
37:     virtual void OnComputerReset() = 0; // Called during Computer Reset
38:     virtual void OnProgramReset() = 0; // Called during Program Reset
39:
40: protected:
41:     bool DoesProgramReset;                      // true if this is reset by P.R. button
42:
43: public:
44:     TCpuObject *NextReset;
45: };
46:
47:
48: //
49: // A second abstract class of objects that not only react to the Resets,
50: // but also have entries on the display panel.
51: //
52:
53: class TDisplayObject : public TCpuObject {
54:
55: public:
56:     TDisplayObject();                            // Constructor to init data.
57:     virtual void Display() = 0;                  // Called to display this item
58:     virtual void LampTest(bool b) = 0; // Called to start/end lamp test
59:
60: public:
61:     TDisplayObject *NextDisplay;
62: };
63:
64: □
65:
66: // Class TDisplayObjects are indicators: They just
```

```
67: // display other things. As a result, they need a pointer to
68: // a function returning bool in order to decide what to do.
69:
70: class TDisplayIndicator : public TDisplayObject {
71:
72: protected:
73:     TLabel *lamp;
74:     bool (_closure *display)();
75:
76: public:
77:
78:     // The constructor requires a pointer to a lamp and a pointer to
79:     // a function that can calculate lamp state.
80:
81:     // We use a closure so that we don't have to pass a pointer to the
82:     // stuff the display function needs (an object pointer is automatically
83:     // embeeded in an _closure *)
84:
85: TDisplayIndicator(TLabel *l,bool (_closure *func)() ) {
86:     lamp = l;
87:     display = func;
88: }
89:
90: virtual void OnComputerReset() { ; }      // These have no state to reset
91: virtual void OnProgramReset() { ; }        // These have no state to reset
92:
93: void Display() {
94:     lamp -> Enabled = display();
95:     lamp -> Repaint();
96: }
97:
98: void LampTest(bool b) {
99:     lamp -> Enabled = (b ? true : display());
100:    lamp -> Repaint();
101: }
102: };
103:
104: □
105:
106: //
107: // Class of TDisplayObjects that are latches:
108: // that can be set, reset and their state read out. Some are
109: // reset by Program Reset (PR) some are not.
110: //
111:
112: class TDisplayLatch : public TDisplayObject {
113:
114: protected:
115:     bool state;                      // Latches can be set or reset
116:     bool doprogramreset;             // Some are reset by PR, some are not.
117:     TLabel *lamp;                  // Pointer to display lamp.
118:
119: public:
120:     TDisplayLatch(TLabel *l);        // Constructor - Set up lamp
121:     TDisplayLatch(TLabel *l, bool progreset); // Same, but inhibit PR
122:
123:     virtual void OnComputerReset(); // Define Computer Reset behavior now
124:     virtual void OnProgramReset(); // Define Program Reset behavior now too
125:
126:     void Display();                // Define display behavior
127:     void LampTest(bool b);         // Define lamp test behavior.
128:
129:     // All you can really do with latches is set/reset/test them
130:
131:     inline void Reset() { state = false; }
132:     inline void Set() { state = true; }
```

```
133:     inline void Set(bool b) { state = b; }
134:     void SetStop(char *msg);
135:     inline bool State() { return state; }
136: };
137:
138: □
139:
140: class TRingCounter : public TDisplayObject {
141: private:
142:     char state;                      // Override "state" variable !!
143:     char max;                        // Max number of entries
144:     TLabel *lastlamp;              // Last lamp to be displayed
145:     TLabel *lastlampCE;             // Last CE lamp to be displayed
146:
147: public:
148:     TLabel **lamps;                // Ptr to array of lamps
149:     TLabel **lampsCE;               // Ptr to array of CE lamps (or 0)
150:
151: public:
152:     TRingCounter(char n);          // Construct with # of entries
153:                                     // Ring counters are always reset by PR
154:
155:     virtual __fastcall ~TRingCounter(); // Destructor for array of lamps
156:
157: // Functions inherited from abstract base classes now need definition
158:
159: void OnComputerReset();
160: void OnProgramReset();
161: void Display();
162: void LampTest(bool b);
163:
164: // The real meat of the Ring Counter class
165:
166: inline void Reset() { state = 0; }
167: inline char Set(char n) { return state = n; }
168: inline char State() { return state; }
169: char Next();
170: };
171:
172: □
173:
174: // Data Registers. All are stored as BCD, but we have special
175: // set routines to set or clear special parts for those registers
176: // that don't use all the bits (e.g. the Op register has no WM or
177: // C bits.
178:
179: // Also, a Register can have an optional pointer to an error latch.
180: // If so, then whenever the register is used, a parity check for ODD
181: // parity is made, and if invalid, the error latch is set. (This
182: // includes use during assignment).
183:
184: class TRegister : public TDisplayObject {
185:
186: private:
187:
188:     BCD value;
189:     TLabel *lamerER;               // If set, there is an error lamp
190:     TLabel **lamps;                // If set, they point to lamps for WM C B A 8 4 2 1
191:                                     // Any lamp not present MUST be set to 0
192:
193: public:
194:
195:     TRegister() { value = BITC; DoesProgramReset = true; lampER = 0; lamps = 0; }
196:     TRegister(bool b) { value = BITC; DoesProgramReset = b; lampER = 0; lamps =
197:         0; }
198:     TRegister(int i) { value = i; DoesProgramReset = true; lampER = 0; lamps = 0;
```

```
    }
198:     TRegister(int i, bool b) { value = i; DoesProgramReset = b; lampER = 0; lamps
= 0; }
199:
200:     inline void OnComputerReset() { Reset(); }
201:     inline void Reset() { value = BITC; }
202:
203:     inline void Set(BCD bcd) { value = bcd; }
204:
205:     inline BCD Get() { return value; }
206:
207:     void operator=(TRegister &source);
208:     void Display();
209:     void LampTest(bool b);
210:
211: // To set up a register to display, provide a pointer to an error
212: // lamp (if any), and an array of pointers to data lamps. Data lamps
213: // for any given bit (e.g. WM) may or may not exist. Order of lamps
214: // is 1 2 4 8 A B C WM (0 thru 7)
215:
216:     void SetDisplay(TLabel *ler, TLabel **l) {
217:         lampER = ler;
218:         lamps = l;
219:     }
220:
221:     void OnProgramReset() {
222:         if(DoesProgramReset) {
223:             Reset();
224:         }
225:     }
226: };
227: □
228: // Address Registers. For efficiency, we keep both binary and
229: // the real 2-out-of-5 code representations. If either one is
230: // valid, and the other representation is requested, we convert
231: // on the fly (and mark that representation valid).
232:
233: // The Set() function effectively implements the Address Channel.
234:
235: class TAddressRegister : public TCpuObject {
236:
237: private:
238:
239:     long i_value;           // Integer equivalent of register
240:     bool i_valid;          // True if integer rep. is valid
241:     bool set[5];           // True if corresponding digit is set
242:     TWOOF5 digits[5];      // Original 2 out of 5 code representation
243:     bool d_valid;          // True if digit rep. is valid
244:     char *name;
245:
246: public:
247:
248:     void OnComputerReset() { Reset(); };
249:     void OnProgramReset() { };
250:
251:     TAddressRegister();    // Constructor / initialization
252:     bool IsValid();        // Returns true if all digits set
253:     long Gate();           // Returns integer value if valid, -1 if not.
254:     BCD GateBCD(int i);   // Returns a single digit
255:     void Set(TWOOF5 digit,int index); // Sets a digit
256:     void Set(long value); // Sets whole register from binary (address mod)
257:     void Reset();          // Resets the register to blanks
258:
259:     void operator=(TAddressRegister &source); // Assignment
260:
261: };
```

```
262:
263: □
264:
265: // This class defines the A Channel - basically, it defines what
266: // register is selected when the A channel is accessed.
267:
268: class TAChannel : public TDisplayObject {
269:
270: public:
271:
272:     enum AChannelSelect { A_Channel_None = 0, A_Channel_A = 1,
273:                           A_Channel_Mod = 2, A_Channel_E = 3, A_Channel_F = 4 };
274:
275: private:
276:
277:     enum AChannelSelect AChannelSelect;
278:     TLabel *lamps[4];
279:
280: public:
281:
282:     TAChannel();                                // Constructor
283:
284:     void OnComputerReset() {
285:         Reset();
286:     }
287:
288:     void OnProgramReset() {
289:         Reset();
290:     }
291:
292:     inline BCD Select(enum AChannelSelect sel); // Select input to A Channel
293:     inline BCD Select();                      // Use whatever was last selected
294:
295:     inline void Reset() { AChannelSelect = A_Channel_None; }
296:
297:     void Display();
298:     void LampTest(bool b);
299: };
300:
301: □
302:
303: // The Assembly Channel: The 1410 Mix-Master!
304:
305: class TAssemblyChannel : public TDisplayObject {
306:
307: public:
308:
309:     enum AsmChannelZonesSelect {
310:         AsmChannelZonesNone = 0, AsmChannelZonesB = 1, AsmChannelZonesA = 2
311:     };
312:
313:     enum AsmChannelWMSelect {
314:         AsmChannelWMNone = 0, AsmChannelWMB = 1, AsmChannelWMA = 2
315:     };
316:
317:     enum AsmChannelNumericSelect {
318:         AsmChannelNumNone = 0, AsmChannelNumB = 1, AsmChannelNumA = 2,
319:         AsmChannelNumAdder = 3
320:     };
321:
322:     enum AsmChannelSignSelect {
323:         AsmChannelSignNone, AsmChannelSignB = 1, AsmChannelSignA = 2,
324:         AsmChannelSignLatch = 3
325:     };
326:
327: private:
```

```
328:
329:     BCD value;           // We remember value, for efficiency
330:     bool valid;          // True if value is set. Reset each cycle!
331:
332:     enum AsmChannelZonesSelect AsmChannelZonesSelect;
333:     enum AsmChannelWMSelect AsmChannelWMSelect;
334:     enum AsmChannelNumericSelect AsmChannelNumericSelect;
335:     enum AsmChannelSignSelect AsmChannelSignSelect;
336:     bool AsmChannelInvertSign;
337:
338:     TLabel *AssmLamps[8];
339:     TLabel *AssmComplLamps[8];
340:     TLabel *AssmERLamp;
341:
342: public:
343:
344:     TAssemblyChannel();
345:
346:     void OnComputerReset() { Reset(); }
347:     void OnProgramReset() { Reset(); }
348:
349:     void Display();
350:     void LampTest(bool b);
351:
352:     BCD Select();           // Uses last value and state
353:
354:     BCD Select(
355:         enum AsmChannelWMSelect WMSelect,
356:         enum AsmChannelZonesSelect ZoneSelect,
357:         bool InvertSign,
358:         enum AsmChannelSignSelect SignSelect,
359:         enum AsmChannelNumericSelect NumSelect
360:     );
361:
362:     BCD Get();
363:
364:     BCD Set(BCD v) { value = v; valid = true; }
365:
366:     void Reset();
367:
368:     BCD GateAChannelToAssembly(BCD v);
369:
370:     BCD GateBRegToAssembly(BCD v);
371:
372: };
373:
374: □
375:
376: // This class defines what is in an I/O Channel
377:
378: #define IOCHNOTREADY    1
379: #define IOCHBUSY        2
380: #define IOCHDATACHECK   4
381: #define IOCHCONDITION   8
382: #define IOCHNOTTRANSFER 16
383: #define IOCHWLRECORD    32
384:
385: #define IOLAMPNOTREADY  0
386: #define IOLAMPBUSY      1
387: #define IOLAMPDATACHECK 2
388: #define IOLAMPCONDITION 3
389: #define IOLAMPNOTTRANSFER 4
390: #define IOLAMPWLRECORD  5
391:
392: class T1410Channel : public TDisplayObject {
393:
```

```
394: public:
395:
396:     // Functions inherited from abstract base class classes now need definition
397:
398:     void OnComputerReset();
399:     void OnProgramReset();
400:     void Display();
401:     void LampTest(bool b);
402:
403: private:
404:
405:     // Channel information
406:
407:     int ChStatus;                                // Channel status (see defines)
408:
409:     TLabel *ChStatusDisplay[6];                  // Channel status lights
410:
411: public:
412:
413:     TRegister *ChOp;
414:     TRegister *ChUnitType;
415:     TRegister *ChUnitNumber;
416:     TRegister *ChR1, *ChR2;
417:
418:     TDisplayLatch *ChInterlock;
419:     TDisplayLatch *ChRBCTInterlock;
420:     TDisplayLatch *ChRead;
421:     TDisplayLatch *ChWrite;
422:     TDisplayLatch *ChOverlap;
423:     TDisplayLatch *ChNotOverlap;
424:
425:     enum TapeDensity {
426:         DENSITY_200_556 = 0, DENSITY_200_800 = 1, DENSITY_556_800 = 2
427:     } TapeDensity;
428:
429:
430:     // Methods
431:
432:     T1410Channel(                                     // Constructor
433:         TLabel *LampInterlock,
434:         TLabel *LampRBCTInterlock,
435:         TLabel *LampRead,
436:         TLabel *LampWRIte,
437:         TLabel *LampOverlap,
438:         TLabel *LampNotOverlap,
439:         TLabel *LampNotRead,
440:         TLabel *LampBusy,
441:         TLabel *LampDataCheck,
442:         TLabel *LampCondition,
443:         TLabel *LampWLRecord,
444:         TLabel *LampNoTransfer
445:     );
446:
447:     inline int SetStatus(int i) { return ChStatus = i; }
448:     inline int GetStatus() { return ChStatus; }
449: };
450:
451:
452: □
453: // This class defines what is actually inside the CPU.
454:
455: #define MAXCHANNEL 2
456: #define CHANNEL1 0
457: #define CHANNEL2 1
458:
459: #define STORAGE 80000
```

```
460:  
461: #define I_RING_OP 0  
462: #define I_RING_1 1  
463: #define I_RING_2 2  
464: #define I_RING_3 3  
465: #define I_RING_4 4  
466: #define I_RING_5 5  
467: #define I_RING_6 6  
468: #define I_RING_7 7  
469: #define I_RING_8 8  
470: #define I_RING_9 9  
471: #define I_RING_10 10  
472: #define I_RING_11 11  
473: #define I_RING_12 12  
474:  
475: #define A_RING_1 0  
476: #define A_RING_2 1  
477: #define A_RING_3 2  
478: #define A_RING_4 3  
479: #define A_RING_5 4  
480: #define A_RING_6 5  
481:  
482: #define CLOCK_A 0  
483: #define CLOCK_B 1  
484: #define CLOCK_C 2  
485: #define CLOCK_D 3  
486: #define CLOCK_E 4  
487: #define CLOCK_F 5  
488: #define CLOCK_G 6  
489: #define CLOCK_H 7  
490: #define CLOCK_J 8  
491: #define CLOCK_K 9  
492:  
493: #define SCAN_N 0  
494: #define SCAN_1 1  
495: #define SCAN_2 2  
496: #define SCAN_3 3  
497:  
498: #define SUB_SCAN_NONE 0  
499: #define SUB_SCAN_U 1  
500: #define SUB_SCAN_B 2  
501: #define SUB_SCAN_E 3  
502: #define SUB_SCAN_MQ 4  
503:  
504: #define CYCLE_A 0  
505: #define CYCLE_B 1  
506: #define CYCLE_C 2  
507: #define CYCLE_D 3  
508: #define CYCLE_E 4  
509: #define CYCLE_F 5  
510: #define CYCLE_I 6  
511: #define CYCLE_X 7  
512:  
513: class T1410CPU {  
514:  
515: private:  
516:  
517:     BCD core[STORAGE];  
518:  
519: public:  
520:  
521:     // Wiring list  
522:  
523:     TCpuObject *ResetList;           // List of latches.  
524:     TDisplayObject *DisplayList;    // List of displayable things  
525:
```

```
526: // Data Registers
527:
528: TRegister *A_Reg, *B_Reg, *Op_Reg, *Op_Mod_Reg;
529:
530: // Address Registers
531:
532: TAddressRegister *STAR; // Storage Address Register
533: // AKA MAR (Memory Address Register)
534:
535: TAddressRegister *A_AR, *B_AR, *C_AR, *D_AR, *E_AR, *F_AR;
536:
537: TAddressRegister *I_AR; // Instruction Counter
538:
539:
540: // Channels
541:
542: T1410Channel *Channel[MAXCHANNEL]; // 2 I/O Channels.
543:
544: TAChannel *AChannel; // A Channel
545: TAssemblyChannel *AssemblyChannel; // Assembly Channel
546:
547: // Indicators
548:
549: TDisplayIndicator *OffNormal; // OFF NORMAL Indicator
550:
551: // Ring Counters
552:
553: TRingCounter *IRing; // Instruction decode ring
554: TRingCounter *ARing; // Address decode ring
555: TRingCounter *ClockRing; // Cycle Clock
556: TRingCounter *ScanRing; // Address Modification Mode
557: TRingCounter *SubScanRing; // Arithmetic Scan type
558: TRingCounter *CycleRing; // CPU Cycle type
559:
560: // Latches with Indicators
561:
562: TDisplayLatch *CarryIn; // Carry latch
563: TDisplayLatch *CarryOut; // Adder has generated carry
564: TDisplayLatch *AComplement; // A channel complement
565: TDisplayLatch *BComplement; // B channel complement
566: TDisplayLatch *CompareBGTA; // B > A
567: TDisplayLatch *CompareBEQA; // B = A
568: TDisplayLatch *CompareBLTA; // B < A NOTE: On after C. Reset.
569: TDisplayLatch *Overflow; // Arithmetic Overflow
570: TDisplayLatch *DivideOverflow; // Divide Overflow
571: TDisplayLatch *ZeroBalance; // Zero arithmetic result
572:
573: // Check Latches
574:
575: TDisplayLatch *AChannelCheck; // A Channel parity error
576: TDisplayLatch *BChannelCheck; // B Channel parity error
577: TDisplayLatch *AssemblyChannelCheck; // Assembly Channel parity error
578: TDisplayLatch *AddressChannelCheck; // Address Channel parity error
579: TDisplayLatch *AddressExitCheck; // Validity error at address reg.
580: TDisplayLatch *ARegisterSetCheck; // A register failed to reset
581: TDisplayLatch *BRegisterSetCheck; // B register failed to reset
582: TDisplayLatch *OpRegisterSetCheck; // Op register failed to set
583: TDisplayLatch *OpModifierSetCheck; // Op modifier failed to set
584: TDisplayLatch *ACharacterSelectCheck; // Incorrect A channel gating
585: TDisplayLatch *BCharacterSelectCheck; // Incorrect B channel geting
586:
587: TDisplayLatch *IOInterlockCheck; // Program did not check I/O
588: TDisplayLatch *AddressCheck; // Program gave bad address
589: TDisplayLatch *RBCInterlockCheck; // Program did not check RBC
590: TDisplayLatch *InstructionCheck; // Program issued invalide op
591:
```

```
592:     // Switches
593:
594:     enum Mode {                                     // Mode switch, values must match
595:         MODE_RUN = 0, MODE_DISPLAY = 1, MODE_ALTER = 2,
596:         MODE_CE = 3, MODE_IE = 4, MODE_ADDR = 5
597:     } Mode;
598:
599:     enum AddressEntry {
600:         ADDR_ENTRY_I = 0, ADDR_ENTRY_A = 1, ADDR_ENTRY_B = 2,
601:         ADDR_ENTRY_C = 3, ADDR_ENTRY_D = 4, ADDR_ENTRY_E = 5,
602:         ADDR_ENTRY_F = 6
603:     } AddressEntry;
604:
605:     // The entry below is for the state of the 1415 Storage Scan SWITCH
606:     // (The storage scan modification mode is in the Ring ScanRing)
607:
608:     enum StorageScan {
609:         SSCAN_OFF = 0, SSCAN_LOAD_1 = 1, SSCAN_LOAD_0 = 2,
610:         SSCAN_REGEN_0 = 3, SSCAN_REGEN_1 = 4
611:     } StorageScan;
612:
613:     enum CycleControl {
614:         CYCLE_OFF = 0, CYCLE_LOGIC = 1, CYCLE_STORAGE = 2
615:     } CycleControl;
616:
617:     enum CheckControl {
618:         CHECK_STOP = 0, CHECK_RESTART = 1, CHECK_RESET = 2
619:     } CheckControl;
620:
621:     bool DiskWrInhibit;
622:     bool AsteriskInsert;
623:     bool InhibitPrintOut;
624:
625:     BCD BitSwitches;
626:
627:     // CPU state latches
628:
629:     bool StopLatch;                                // True to stop CPU
630:     bool StopKeyLatch;                            // True if STOP button pressed
631:     bool DisplayModeLatch;                      // True if we are displaying storage
632:     bool ProcessRoutineLatch;                    // True if we are in Run or IE mode
633:     bool BranchLatch;                           // True if we are to branch
634:     bool BranchTo1Latch;                        // True to branch to 1
635:     bool LastInstructionReadout;                // True at end of Instruction fetch
636:     bool IRingControl;                          // Set to start Instruction Fetch
637:
638:     bool IOMoveModeLatch;                       // I/O Latches
639:     bool IOLoadModeLatch;                      // to check for I/O Overlap
640:
641:     bool StorageWrapLatch;
642:
643:     int IOChannelSelect;                         // One if if channel 2 op.
644:
645:     // Index latches are rolled up into an int. The BA zones from the
646:     // tens and hundreds positions are shifted and the 4 bits together
647:     // generate a value from 0 to 15.
648:
649:     int IndexLatches;
650:
651:     // Methods
652:
653:     T1410CPU();                                 // Constructor
654:     void Display();                            // Run thru the display list
655:     void Cycle();                             // Used for common CPU Cycles
656:
657:     // The 1410 Adder accepts BCD inputs, a Carry Latch and complement
```

```
658:     // flags and returns a sum in BCD, possible setting Carry Out.
659:
660:     BCD Adder(BCD A,int Complement_A,BCD B,int Complement_B);
661:     BCD AdderResult;                                // For the Assembly Channel
662:
663: private:
664:
665:     // Indicator Routines
666:     // Normally, these will be accessed via a bool (_closure *func)()
667:
668:     bool IndicatorOffNormal();
669:
670: public:
671:
672:     // Core operations (using STAR/MAR and B Data Register)
673:
674:     void Readout();           // Reads out one storage character
675:     void Store();            // Stores character in B Data Register
676:     void Store(BCD bcd);     // Sets B Data Register, then stores
677:     void SetScan(char s);   // Sets Scan Modification value
678:
679:     long STARScan();         // Applies Scan CTRL modification to STAR,
680:                             // and returns the results - suitable to assign
681:
682:     long STARMOD(int mod);  // Similar, but provides direct +1/0/-1 mod
683:
684:     void LoadCore(char *file); // Loads core from a file
685:     void DumpCore(char *file); // Dumps core to a file
686:
687: public:
688:
689:     // Instruction operations
690:
691:     unsigned short OpReadOutLines;
692:     unsigned short OpOperationalLines;
693:     unsigned short OpControlLines;
694:
695:     void InstructionDecodeStart(); // Starts instruction decode processing
696:     void InstructionDecode();    // Remainder of instruction decode
697:     void InstructionDecodeIARAdvance(); // Conditionally advance IAR
698:
699:     void InstructionIndexStart(); // Starts up indexing operation
700:     void InstructionIndex();   // Does the actual indexing
701:
702:
703: };
704:
705: extern T1410CPU *CPU;
706:
707: //-----
708: #endif
709:
```

```
1: // This Unit provides the functionality behind some of my private
2: // types for the 1410 emulator
3:
4: //-----
5: #include <vcl\vcl.h>
6: #include <assert.h>
7: #pragma hdrstop
8:
9: #include "UI1410CPU.h"
10: #include "UI1415L.h"
11: #include "UI1415CE.h"
12: #include "UI1410PWR.h"
13: #include "UI1410DEBUG.h"
14: #include "UI1410CPU.h"
15:
16: //-----
17:
18: // Declarations for Borland VCL controls
19:
20: #include <vcl\Classes.hpp>
21: #include <vcl\Controls.hpp>
22: #include <vcl\StdCtrls.hpp>
23:
24: // Core pre-load (gives this virtual 1410 a 7010-like load capability
25: // Just hit Computer Reset then Start to boot from tape
26:
27: char *core_load_chars = "AL%B000012$N..";
28: char core_load_wm[] = { 0,1,0,0,0,0,0,0,0,0,1,1,1 };
29:
30: // Tables which pre-multiply integers by decimal numbers, for efficiency.
31:
32: long ten_thousands[] = {
33:     0,10000,20000,30000,40000,50000,60000,70000,80000,90000
34: };
35:
36: long thousands[] = {
37:     0,1000,2000,3000,4000,5000,6000,7000,8000,9000
38: };
39:
40: long hundreds[] = {
41:     0,100,200,300,400,500,600,700,800,900
42: };
43:
44: long tens[] = { 0,10,20,30,40,50,60,70,80,90 };
45:
46: // Storage Scan modification values
47: // Correspond to the values in the enum StorageScan
48:
49: long scan_mod[] = { 0, -1, +1, -1 };
50:           N | 2   3
51: // Tables for processing signs (shifted right 4 bits!)
52:
53: // Table to take a sign bit configuration (BA) and normalize it.
54: // Plus sign is normalized to BA, Minus sign is normalized to just A
55:
56: unsigned char sign_normalize_table[] = { 3, 3, 2, 3 };
57:
58: // Same thing, but inverts the sign in the process
59:
60: unsigned char sign_complement_table[] = { 2, 2, 3, 2 };
61:
62: // Same thing, but indicates negative by 1
63:
64: unsigned char sign_negative_table[] = { 0, 0, 1, 0 };
65:
66: // Implementation of TCpuObject (Abstract Base Class)
```

```
67:
68: // Everything in the CPU is on the reset list.
69: // Everything is reset by Computer Reset
70: // NOT Everything is reset by Program Reset!
71:
72: TCpuObject::TCpuObject()
73: {
74:     NextReset = CPU -> ResetList;
75:     CPU -> ResetList = this;
76: }
77:
78: // Implementation of TDisplayObject (Abstract Base Class)
79:
80: // What is special about these objects is that they are on the display list.
81:
82: TDisplayObject::TDisplayObject()
83: {
84:     NextDisplay = CPU -> DisplayList;
85:     CPU -> DisplayList = this;
86: }
87:
88: □
89:
90: // Implementation of TDisplayLatch (Display Latch Base Class)
91:
92: // The constructor initializes the latch and sets the pointer to a lamp.
93: // A pointer to a lamp is required.
94:
95: // Default is to be reset by a Program Reset, but this can be overridden
96: // when the constructor is called, if necessary.
97:
98: TDisplayLatch::TDisplayLatch(TLabel *l)
99: {
100:     state = false;
101:     doprogramreset = true;
102:     lamp = l;
103: }
104:
105: // The second constructor does the same thing, but is passed a variable
106: // which indicates whether or not the latch should be reset by Program
107: // Reset.
108:
109: TDisplayLatch::TDisplayLatch(TLabel *l, bool progreset)
110: {
111:     state = false;
112:     doprogramreset = progreset;
113:     lamp = l;
114: }
115:
116: // All Displayable Latches are reset on a COMPUTER RESET
117:
118: void TDisplayLatch::OnComputerReset()
119: {
120:     Reset();
121: }
122:
123: // Whether or not a displayable latch is reset by program reset depends
124: // on the latch.
125:
126: void TDisplayLatch::OnProgramReset()
127: {
128:     if(doprogramreset) {
129:         Reset();
130:     }
131: }
132:
```

```
133: // Routine to set a latch and set the CPU Stop Latch at the same time
134: // Typically used for error latches.
135:
136: void TDisplayLatch::SetStop(char *msg) {
137:     state = true;
138:     CPU -> StopLatch = true;
139:     if(msg != 0) {
140:         F1410Debug -> DebugOut(msg);
141:     }
142: }
143:
144: // When the Display routine is called, it sets or resets the lamp,
145: // depending on the current state.
146:
147: void TDisplayLatch::Display()
148: {
149:     lamp -> Enabled = state;
150:     lamp -> Repaint();
151: }
152:
153: // On a lamp test, light all the lamp.
154: // On reset of a lamp test, display the current state.
155:
156: void TDisplayLatch::LampTest(bool b)
157: {
158:     lamp -> Enabled = (b ? true : state);
159:     lamp -> Repaint();
160: }
161:
162: □
163:
164: // Implementation of TRingCounter (Ring Counter Class)
165:
166: // The constructor sets the max state of the ring, and resets the ring
167: // It also allocates an array of pointers to the lamps. The creator must
168: // fill in that array, however. Initially, the lamp pointers are empty,
169: // (which means no lamp is attached to that state).
170:
171: TRingCounter::TRingCounter(char n)
172: {
173:     int i;
174:
175:     state = 0;
176:     max = n;
177:     lastlamp = 0;
178:     lastlampCE = 0;
179:     lampsCE = 0;
180:
181:     lamps = new TLabel*[n];
182:     for(i=0; i < n; ++i) {
183:         lamps[i] = 0;
184:     }
185: }
186:
187: // The destructor frees up the array of lamps. Probably will never use.
188:
189: _fastcall TRingCounter::~TRingCounter()
190: {
191:     delete[] lamps;
192:     if(lampsCE != 0) {
193:         delete[] lampsCE;
194:     }
195: }
196:
197: // All Ring counters are reset on PROGRAM and COMPUTER RESET
198:
```

```
199: void TRingCounter::OnComputerReset()
200: {
201:     state = 0;
202: }
203:
204: void TRingCounter::OnProgramReset()
205: {
206:     state = 0;
207: }
208:
209: // When the Display routine is called, it resets the lamp corresponding
210: // to the state when it last displayed, and then displays the current state
211: // If there is no lamp associated with a state (lamp pointer is null),
212: // then don't display any lamp.
213:
214: void TRingCounter::Display()
215: {
216:     if(lastlamp) {
217:         lastlamp -> Enabled = false;
218:         lastlamp -> Repaint();
219:     }
220:     if(lamps[state] == 0) {
221:         lastlamp = 0;
222:         return;
223:     }
224:     lamps[state] -> Enabled = true;
225:     lamps[state] -> Repaint();
226:     lastlamp = lamps[state];
227:
228:     if(lampsCE != 0) {
229:         if(lastlampCE) {
230:             lastlampCE -> Enabled = false;
231:             lastlampCE -> Repaint();
232:         }
233:         if(lampsCE[state] == 0) {
234:             lastlampCE = 0;
235:             return;
236:         }
237:         lampsCE[state] -> Enabled = true;
238:         lampsCE[state] -> Repaint();
239:         lastlampCE = lampsCE[state];
240:     }
241: }
242:
243: // On a lamp test, light all the associated lamps.
244: // On reset of a lamp test, clear them all, then display the current state.
245:
246: void TRingCounter::LampTest(bool b)
247: {
248:     int i;
249:
250:     for(i = 0; i < max; ++i) {
251:         if(lamps[i] != 0) {
252:             lamps[i] -> Enabled = b;
253:             lamps[i] -> Repaint();
254:         }
255:         if(lampsCE != 0 && lampsCE[i] != 0) {
256:             lampsCE[i] -> Enabled = b;
257:             lampsCE[i] -> Repaint();
258:         }
259:     }
260:     if(!b) {
261:         lastlamp = 0;
262:         lastlampCE = 0;
263:         this -> Display();
264:     }
}
```

```
265: }
266:
267: // Next advances to the next state (or back to the start if appropriate)
268: // Returns current state. This is only useful for true Ring counters.
269:
270: char TRingCounter::Next()
271: {
272:     if(++state >= max) {
273:         state = 0;
274:     }
275:     return(state);
276: }
277:
278: □
279:
280: // Implementation of Simple Registers
281:
282: // Assignment: Just assign the value, NOT the things from TCpuObject!!
283: // (Those need to stay unchanged as they should be invariant once the
284: // register is created: it's position on the reset list and whether or
285: // not it is affected by Program Reset, for example.
286:
287: void TRegister::operator=(TRegister &source)
288: {
289:     value = source.value;
290: }
291:
292: // Display and LampTest. These only do anything if the display variables
293: // are actually set.
294:
295: void TRegister::Display()
296: {
297:     int bitmask = 0x1;
298:     int i;
299:
300:     for(i=0; i < 8; ++i) {
301:         if(lamps != 0 && lamps[i] != 0) {
302:             lamps[i] -> Enabled = ((value.ToInt() & bitmask) != 0);
303:             lamps[i] -> Repaint();
304:         }
305:         bitmask <= 1;
306:     }
307:
308:     if(lampER != 0) {
309:         lampER -> Enabled = (!value.CheckParity());
310:         lampER -> Repaint();
311:     }
312: }
313:
314: void TRegister::LampTest(bool b)
315: {
316:     int i;
317:
318:     if(b) {
319:         for(i=0; i < 8; ++i) {
320:             if(lamps != 0 && lamps[i] != 0) {
321:                 lamps[i] -> Enabled = true;
322:                 lamps[i] -> Repaint();
323:             }
324:         }
325:         if(lampER != 0) {
326:             lampER -> Enabled = true;
327:             lampER -> Repaint();
328:         }
329:     }
330: else {
```

```
331:         Display();
332:     }
333: }
334:
335: □
336:
337: // Implementation of Address Registers
338:
339: // The constructor just initializes things so that we know that the
340: // address register contains an invalid value.
341:
342: TAddressRegister::TAddressRegister()
343: {
344:     i_valid = false;
345:     i_value = 0;
346:     d_valid = false;
347:     set[0] = set[1] = set[2] = set[3] = set[4] = false;
348:     DoesProgramReset = false;
349: }
350:
351: // A function to determine whether or not the address register contains
352: // a valid value.
353:
354: bool TAddressRegister::IsValid()
355: {
356:     if(i_valid || d_valid ||
357:         (set[0] && set[1] && set[2] && set[3] && set[4]) ) {
358:         return(true);
359:     }
360:     return(false);
361: }
362:
363: // A routine to reset an address register: to binary 0. Note that this
364: // means it is invalid (i.e. will cause an Address Exit Check if you try
365: // and actually use the value except to print it out on the console.
366:
367: void TAddressRegister::Reset()
368: {
369:     TWOOF5 zero;
370:
371:     i_valid = false;
372:     d_valid = false;
373:     set[0] = set[1] = set[2] = set[3] = set[4] = false;
374:     digits[0] = digits[1] = digits[2] = digits[3] = digits[4] = zero;
375:     i_value = 0;
376: }
377:
378: // A routine to get the value of an address register. Note that if
379: // the value is invalid, the result is an Address Exit Check.
380:
381: long TAddressRegister::Gate()
382: {
383:     if(i_valid) {
384:         return(i_value);
385:     }
386:     else if(IsValid()) {
387:         i_value = ten_thousands[digits[0].ToInt()] +
388:                   thousands[digits[1].ToInt()] +
389:                   hundreds[digits[2].ToInt()] +
390:                   tens[digits[3].ToInt()] + digits[4].ToInt();
391:         i_valid = true;
392:         d_valid = true;
393:         return(i_value);
394:     }
395:     else {
396:         CPU -> AddressExitCheck ->
```

```
397:             SetStop("Address Exit Check during Gate()");
398:             return(-1);
399:         }
400:     }
401:
402: // Get a single character from an address register. This is valid even
403: // if the value in the register is invalid.
404:
405: BCD TAddressRegister::GateBCD(int i)
406: {
407:     if(d_valid) {
408:         if(digits[i-1].ToInt() < 0) {
409:             CPU -> AddressExitCheck ->
410:                 SetStop("Address Exit Check during GateBCD(#1)");
411:                 F1415L -> DisplayAddrChannel(digits[i-1],true);
412:                 return(0);
413:         }
414:         return(digits[i-1].ToBCD());
415:     } i-valid
416:     else if(iValid()) {
417:         digits[4] = i_value % 10;
418:         digits[3] = (i_value % 100)/10;
419:         digits[2] = (i_value % 1000)/100;
420:         digits[1] = (i_value % 10000)/1000;
421:         digits[0] = i_value /10000;
422:         d_valid = true;
423:         set[0] = set[1] = set[2] = set[3] = set[4] = true; |
424:         if(digits[i-1].ToInt() < 0) {
425:             CPU -> AddressExitCheck ->
426:                 SetStop("Address Exit Check during GateBCD(#2)");
427:                 return(0);
428:         }
429:         return(digits[i-1].ToBCD());
430:     }
431:     else {
432:
433:         // Register is not set, but it is still valid to read out a single
434:         // digit. This should not cause an error.
435:
436: /*     CPU -> AddressExitCheck ->
437:         SetStop("Address Exit Check during GateBCD(Contents not valid)");
438:         F1415L -> DisplayAddrChannel(TWOOF5(0),true);
439:         return(BCD(0));
440: */
441:
442:     if(set[i-1] && digits[i-1].ToInt() >= 0) {
443:         return(digits[i-1].ToBCD());
444:     }
445:     else {
446:         return(0);
447:     }
448: }
449: }
450:
451: // A routine to set a single digit of an address register.
452: // This effectively implements the address channel.
453:
454: void TAddressRegister::Set(TWOOF5 digit,int i)
455: {
456:     if(digit.ToInt() == -1) {
457:         CPU -> AddressChannelCheck ->
458:             SetStop("Address Channel Check while setting address register");
459:             F1415L -> DisplayAddrChannel(digit,true);
460:     }
461:
462:     digits[i-1] = digit;
```

*if i-valid == true {*

```
463:     set[i-1] = true;
464:     if(i == 5 && set[0] && set[1] && set[2] && set[3]) {
465:         d_valid = true;
466:     }
467: }
468:
469: // A routine to set an address register from a binary value. Should
470: // really only be used to reset IAR to 1 during a reset operation.
471:
472: void TAddressRegister::Set(long i)
473: {
474:     d_valid = false;
475:     set[0] = set[1] = set[2] = set[3] = set[4] = false;
476:     i_valid = true;
477:     i_value = i;
478: }
479:
480: // Assignment: Just assign the value, NOT the things from TCpuObject!!
481: // (Those need to stay unchanged as they should be invariant once the
482: // register is created: it's position on the reset list and whether or
483: // not it is affected by Program Reset, for example.
484:
485: // Note that an attempt to assign from an invalid register does do the
486: // set, but also sets the Address Exit Check error in the CPU
487:
488: void TAddressRegister::operator=(TAddressRegister &source)
489: {
490:     int i;
491:
492:     i_value = source.i_value;
493:     i_valid = source.i_valid;
494:     d_valid = source.d_valid;
495:     if(!source.IsValid()) {
496:         CPU -> AddressExitCheck ->
497:             SetStop("Address Exit Check assigning - from register invalid");
498:     }
499:     for(i=0; i < 5; ++i) {
500:         digits[i] = source.digits[i];
501:         set[i] = source.set[i];
502:     }
503: }
504:
505: □
506:
507: // Implementation of A Channel
508:
509: // Constructor. There is only 1 a channel, so it sets pointers to lamps
510: // as well as initializing
511:
512: TAChannel::TAChannel()
513: {
514:     AChannelSelect = A_Channel_None;
515:
516:     lamps[0] = F1415L -> Light_CE_ACh_A;
517:     lamps[1] = F1415L -> Light_CE_ACh_d;
518:     lamps[2] = F1415L -> Light_CE_ACh_E;
519:     lamps[3] = F1415L -> Light_CE_ACh_F;
520: }
521:
522: // Routine to return what is currently selected on A Channel
523:
524: BCD TAChannel::Select()
525: {
526:     return Select(AChannelSelect);
527: }
528:
```

```
529: // Routine to set what the A Channel will select. It also returns that
530: // selected value, for convenience.
531:
532: BCD TACchannel::Select(enum AChannelSelect sel)
533: {
534:     AChannelSelect = sel;
535:
536:     switch(AChannelSelect) {
537:
538:         case A_Channel_None:
539:             return(0);
540:         case A_Channel_A:
541:             return(CPU -> A_Reg -> Get());
542:         case A_Channel_Mod:
543:             return(CPU -> Op_Mod_Reg -> Get());
544:         case A_Channel_E:
545:             return(CPU -> Channel[CHANNEL1] -> ChR2 -> Get());
546:         case A_Channel_F:
547:             if(MAXCHANNEL > 1) {
548:                 return(CPU -> Channel[CHANNEL2] -> ChR2 -> Get());
549:             }
550:             else {
551:                 CPU -> ACharacterSelectCheck ->
552:                     SetStop("A Character Select Check: Channel 2, only 1
553: configured");
554:             }
555:         default:
556:             CPU -> ACharacterSelectCheck ->
557:                 SetStop("A Character Select Check: Invalid selection");
558:             return(0);
559:     }
560: }
561:
562: // Routine to display the current A Channel Select status
563:
564: void TACchannel::Display()
565: {
566:     int i;
567:
568:     for(i=0; i < 4; ++i) {
569:         lamps[i] -> Enabled = false;
570:     }
571:
572:     switch(AChannelSelect) {
573:
574:         case A_Channel_None:
575:             break;
576:         case A_Channel_A:
577:             lamps[0] -> Enabled = true;
578:             break;
579:         case A_Channel_Mod:
580:             lamps[1] -> Enabled = true;
581:             break;
582:         case A_Channel_E:
583:             lamps[2] -> Enabled = true;
584:             break;
585:         case A_Channel_F:
586:             if(MAXCHANNEL > 1) {
587:                 lamps[3] -> Enabled = true;
588:             }
589:             break;
590:     }
591:
592:     for(i = 0; i < 4; ++i) {
593:         lamps[i] -> Repaint();
```

```
594:     }
595: }
596:
597: // Lamp Test
598:
599: void TAChannel::LampTest(bool b)
600: {
601:     int i;
602:
603:     if(b) {
604:         for(i = 0; i < 4; ++i) {
605:             lamps[i] -> Enabled = true;
606:             lamps[i] -> Repaint();
607:         }
608:     }
609:     else {
610:         Display();
611:     }
612: }
613:
614: □
615:
616: // Implementation of Assembly Channel
617:
618: // Private tables for assembly channel. These are bitmasks. In general,
619: // these arrays have elements in the order of their respective enums.
620: // Each element has 3 entries, one byte each for the A Channel, the B Channel/
621: // Register and the Adder
622:
623: struct AssmMask {
624:     int AChannelMask;
625:     int BChannelMask;
626:     int AdderMask;
627: };
628:
629: static struct AssmMask AssmWMMask[] = {
630:     { 0x00, 0x00, 0x00 },           // No WM
631:     { 0x00, 0x80, 0x00 },           // B WM
632:     { 0x80, 0x00, 0x00 }            // A WM
633: };
634:
635: static struct AssmMask AssmZonesMask[] = {
636:     { 0x00, 0x00, 0x00 },           // No Zones
637:     { 0x00, 0x30, 0x00 },           // B Zones
638:     { 0x30, 0x00, 0x00 }            // A Zones
639: };
640:
641: static struct AssmMask AssmNumMask[] = {
642:     { 0x00, 0x00, 0x00 },           // No Numerics
643:     { 0x00, 0x0f, 0x00 },           // B Numerics
644:     { 0x0f, 0x00, 0x00 },           // A Numerics
645:     { 0x00, 0x00, 0x0f }            // Adder Numerics
646: };
647:
648: static struct AssmMask AssmSignMask[] = {
649:     { 0x00, 0x00, 0x00 },           // No Sign
650:     { 0x00, 0x30, 0x00 },           // B Sign
651:     { 0x30, 0x00, 0x00 },           // A Sign
652:     { 0x00, 0x00, 0x30 }            // Sign Latch
653: };
654:
655: // Constructor
656:
657: TAAssemblyChannel::TAAssemblyChannel()
658: {
659:     AssmLamps[0] = F1415L -> Light_CE_Assm_1;
```

```
660:     AssmLamps[1] = F1415L -> Light_CE_Assm_2;
661:     AssmLamps[2] = F1415L -> Light_CE_Assm_4;
662:     AssmLamps[3] = F1415L -> Light_CE_Assm_8;
663:     AssmLamps[4] = F1415L -> Light_CE_Assm_A;
664:     AssmLamps[5] = F1415L -> Light_CE_Assm_B;
665:     AssmLamps[6] = F1415L -> Light_CE_Assm_C;
666:     AssmLamps[7] = F1415L -> Light_CE_Assm_WM;
667:
668:     AssmComplLamps[0] = F1415L -> Light_CE_Assm_N1;
669:     AssmComplLamps[1] = F1415L -> Light_CE_Assm_N2;
670:     AssmComplLamps[2] = F1415L -> Light_CE_Assm_N4;
671:     AssmComplLamps[3] = F1415L -> Light_CE_Assm_N8;
672:     AssmComplLamps[4] = F1415L -> Light_CE_Assm_NA;
673:     AssmComplLamps[5] = F1415L -> Light_CE_Assm_NB;
674:     AssmComplLamps[6] = F1415L -> Light_CE_Assm_NC;
675:     AssmComplLamps[7] = F1415L -> Light_CE_Assm_NWM;
676:
677:     AssmERLamp = F1415L -> Light_CE_Assm_ER;
678:
679:     Reset();
680: }
681:
682: // Display and Lamptest have to display both positive and complement.
683: // Otherwise, very similar to a TRegister
684:
685: void TAssemblyChannel::Display()
686: {
687:     int bitmask = 0x1;
688:     int i;
689:
690:     for(i=0; i < 8; ++i) {
691:         AssmLamps[i] -> Enabled = ((value.ToInt() & bitmask) != 0);
692:         AssmComplLamps[i] -> Enabled = !AssmLamps[i] -> Enabled;
693:         AssmLamps[i] -> Repaint();
694:         AssmComplLamps[i] -> Repaint();
695:         bitmask <<= 1;
696:     }
697:
698:     AssmERLamp -> Enabled = (!valid || !value.CheckParity());
699:     AssmERLamp -> Repaint();
700: }
701:
702: void TAssemblyChannel::LampTest(bool b)
703: {
704:     int i;
705:
706:     if(b) {
707:         for(i=0; i < 8; ++i) {
708:             AssmLamps[i] -> Enabled = true;
709:             AssmComplLamps[i] -> Enabled = true;
710:             AssmLamps[i] -> Repaint();
711:             AssmComplLamps[i] -> Repaint();
712:         }
713:         AssmERLamp -> Enabled = true;
714:         AssmERLamp -> Repaint();
715:     }
716:     else {
717:         Display();
718:     }
719: }
720:
721: // Select sets the Assembly Channel selection variables and then
722: // returns the result
723:
724: BCD TAssemblyChannel::Select(
725:     enum AsmChannelWMSelect WMSelect,
```

```
726:     enum AsmChannelZonesSelect ZoneSelect,
727:     bool InvertSign,
728:     enum AsmChannelSignSelect SignSelect,
729:     enum AsmChannelNumericSelect NumSelect)
730: {
731:     bool SignLatch = false; // Ditto for sign latch
732:
733:     BCD Zones, WM, Numerics;
734:
735:     // Remember the selection, for next time
736:
737:     AsmChannelWMSelect = WMSelect;
738:     AsmChannelZonesSelect = ZoneSelect;
739:     AsmChannelInvertSign = InvertSign;
740:     AsmChannelSignSelect = SignSelect;
741:     AsmChannelNumericSelect = NumSelect;
742:
743:     // Can't select both sign and zones!
744:
745:     if(AsmChannelZonesSelect != AsmChannelZonesNone &&
746:         AsmChannelSignSelect != AsmChannelSignNone) {
747:         CPU -> AssemblyChannelCheck ->
748:             SetStop("Assembly Channel Check: Selected both Sign and Zones");
749:         return(0);
750:     }
751:
752:     WM =
753:         (CPU -> AChannel -> Select() &
754:          AssmWMMask[AsmChannelWMSelect].AChannelMask) |
755:          (CPU -> B_Reg -> Get() & AssmWMMask[AsmChannelWMSelect].BChannelMask);
756:
757:     Numerics =
758:         (CPU -> AChannel -> Select() &
759:          AssmNumMask[AsmChannelNumericSelect].AChannelMask) |
760:          (CPU -> B_Reg -> Get() &
761:          AssmNumMask[AsmChannelNumericSelect].BChannelMask) |
762:          (CPU -> AdderResult & AssmNumMask[AsmChannelNumericSelect].AdderMask);
763:
764:     if(AsmChannelSignSelect != AsmChannelSignNone) {
765:         if(AsmChannelSignSelect == AsmChannelSignLatch) {
766:             Zones = (SignLatch ? 0x20 : 0x30);
767:         }
768:         else {
769:             Zones =
770:                 (CPU -> AChannel -> Select() &
771:                  AssmSignMask[AsmChannelSignSelect].AChannelMask) |
772:                  (CPU -> B_Reg -> Get() &
773:                  AssmSignMask[AsmChannelSignSelect].BChannelMask);
774:             }
775:             Zones = Zones >> 4; // Prepare to normalize
776:             Zones = BCD( (AsmChannelInvertSign ?
777:                             sign_complement_table[Zones.ToInt()] :
778:                             sign_normalize_table[Zones.ToInt()]) << 4 );
779:         }
780:
781:         value = WM | Zones | Numerics;
782:         value.SetOddParity();
783:         valid = true;
784:         return(value);
```

```
785: }
786:
787: // Same thing, only we use the existing values. If the value is already
788: // valid, we can take a shortcut.
789:
790: BCD TAssemblyChannel::Select()
791: {
792:     if(valid) {
793:         return(value);
794:     }
795:
796:     return(Select(AsmChannelWMSelect,AsmChannelZonesSelect,
797:                     AsmChannelInvertSign,AsmChannelSignSelect,AsmChannelNumericSelect));
798: }
799:
800: // Get really does the same thing as select, except that it flags an
801: // error if there is not a valid value already.
802:
803: BCD TAssemblyChannel::Get()
804: {
805:     if(valid) {
806:         return(value);
807:     }
808:
809:     CPU -> AssemblyChannelCheck ->
810:         SetStop("Assembly Channel Check: Value not valid at this time");
811:     return(0);
812: }
813:
814: // Method to reset the assembly channel
815:
816: void TAssemblyChannel::Reset() {
817:     value = BITC;
818:     valid = true;
819:     AsmChannelWMSelect = AsmChannelWMNone;
820:     AsmChannelZonesSelect = AsmChannelZonesNone;
821:     AsmChannelInvertSign = false;
822:     AsmChannelSignSelect = AsmChannelSignNone;
823:     AsmChannelNumericSelect = AsmChannelNumNone;
824: }
825:
826: // Common operation: Gate A Channel to Assembly Channel
827:
828: BCD TAssemblyChannel::GateAChannelToAssembly(BCD v) {
829:     AsmChannelWMSelect = AsmChannelWMA;
830:     AsmChannelZonesSelect = AsmChannelZonesA;
831:     AsmChannelInvertSign = false;
832:     AsmChannelSignSelect = AsmChannelSignNone;
833:     AsmChannelNumericSelect = AsmChannelNumA;
834:     value = v;           // Usually this will be an AChannel call return val
835:     valid = true;
836:     return(v);
837: }
838:
839: BCD TAssemblyChannel::GateBRegToAssembly(BCD v) {
840:     AsmChannelWMSelect = AsmChannelWMB;
841:     AsmChannelZonesSelect = AsmChannelZonesB;
842:     AsmChannelInvertSign = false;
843:     AsmChannelSignSelect = AsmChannelSignNone;
844:     AsmChannelNumericSelect = AsmChannelNumB;
845:     value = v;           // Usually this will be a B_Reg call return val
846:     valid = true;
847:     return(v);
848: }
849:
850: □
```

```
851:
852: // Implementation of I/O Channel Class
853:
854: // Constructor.  Initializes state
855:
856: T1410Channel::T1410Channel(
857:     TLabel *LampInterlock,
858:     TLabel *LampRBCInterlock,
859:     TLabel *LampRead,
860:     TLabel *LampWrite,
861:     TLabel *LampOverlap,
862:     TLabel *LampNotOverlap,
863:     TLabel *LampNotReady,
864:     TLabel *LampBusy,
865:     TLabel *LampDataCheck,
866:     TLabel *LampCondition,
867:     TLabel *LampWLRecord,
868:     TLabel *LampNoTransfer ) {
869:
870:     ChStatus = 0;
871:     TapeDensity = DENSITY_200_556;
872:
873:     ChStatusDisplay[IOLAMPNOTREADY] = LampNotReady;
874:     ChStatusDisplay[IOLAMPBUSY] = LampBusy;
875:     ChStatusDisplay[IOLAMPDATACHECK] = LampDataCheck;
876:     ChStatusDisplay[IOLAMPCONDITION] = LampCondition;
877:     ChStatusDisplay[IOLAMPNOTTRANSFER] = LampNoTransfer;
878:     ChStatusDisplay[IOLAMPWLRECORD] = LampWLRecord;
879:
880:     // Generally, the channel latches are *not* reset by Program Reset
881:
882:     ChInterlock = new TDisplayLatch(LampInterlock, false);
883:     ChRBCInterlock = new TDisplayLatch(LampRBCInterlock, false);
884:     ChRead = new TDisplayLatch(LampRead, false);
885:     ChWrite = new TDisplayLatch(LampWrite, false);
886:     ChOverlap = new TDisplayLatch(LampOverlap, false);
887:     ChNotOverlap = new TDisplayLatch(LampNotOverlap, false);
888:
889:     // Generally, the channel registers are *not* reset by Program Reset
890:
891:     ChOp = new TRegister(false);
892:     ChUnitType = new TRegister(false);
893:     ChUnitNumber = new TRegister(false);
894:     ChR1 = new TRegister(false);
895:     ChR2 = new TRegister(false);
896: }
897:
898: // Channel is reset during ComputerReset
899:
900: void T1410Channel::OnComputerReset()
901: {
902:     ChStatus = 0;
903:
904:     // Note: The objects which are TDisplayLatch objects will reset themselves
905: }
906:
907: // Channel is not reset during Program Reset
908:
909: void T1410Channel::OnProgramReset()
910: {
911:     // Channel not affected by Program Reset
912: }
913:
914: // Display Routine.
915:
916: void T1410Channel::Display() {
```

```
917:  
918:     int i;  
919:  
920:     ChStatusDisplay[IOLAMPNOTREADY] -> Enabled =  
921:         ((ChStatus & IOCHNOTREADY) != 0);  
922:     ChStatusDisplay[IOLAMPBUSY] -> Enabled =  
923:         ((ChStatus & IOCHBUSY) != 0);  
924:     ChStatusDisplay[IOLAMPDATACHECK] -> Enabled =  
925:         ((ChStatus & IOCHDATACHECK) != 0);  
926:     ChStatusDisplay[IOLAMPCONDITION] -> Enabled =  
927:         ((ChStatus & IOCHCONDITION) != 0);  
928:     ChStatusDisplay[IOLAMPWLRECORD] -> Enabled =  
929:         ((ChStatus & IOCHWLRECORD) != 0);  
930:     ChStatusDisplay[IOLAMPNOTTRANSFER] -> Enabled =  
931:         ((ChStatus & IOCHNOTTRANSFER) != 0);  
932:  
933:     for(i=0; i <= 5; ++i) {  
934:         ChStatusDisplay[i] -> Repaint();  
935:     }  
936:  
937: // Although in most instances the following would be redundant,  
938: // because these objects are also on the CPU display list, we include  
939: // them here in case we want to display a channel separately.  
940:  
941:     ChInterlock -> Display();  
942:     ChRBCInterlock -> Display();  
943:     ChRead -> Display();  
944:     ChWrite -> Display();  
945:     ChOverlap -> Display();  
946:     ChNotOverlap -> Display();  
947: }  
948:  
949: // Channel Lamp Test  
950:  
951: void T1410Channel::LampTest(bool b)  
952: {  
953:     int i;  
954:  
955: // Note, we don't have to do anything to the TDisplayLatch objects in  
956: // the channel for lamp test. They will take care of themselves on a  
957: // lamp test.  
958:  
959:     if(!b) {  
960:         for(i=0; i <= 5; ++i) {  
961:             ChStatusDisplay[i] -> Enabled = true;  
962:             ChStatusDisplay[i] -> Repaint();  
963:         }  
964:     }  
965:     else {  
966:         Display();  
967:     }  
968: }  
969:  
970: □  
971:  
972: // CPU object constructor. Essentially this method "wires" the 1410.  
973:  
974: T1410CPU::T1410CPU()  
975: {  
976:     long i;  
977:     TLabel *tmpcr, **tmpl;  
978:  
979:     CPU = this;  
980:  
981:     // Clear out the lists  
982:
```

```
983:     DisplayList = 0;
984:     ResetList = 0;
985:
986:     // Set switches to initial states
987:
988:     Mode = MODE_RUN;
989:     AddressEntry = ADDR_ENTRY_I;
990:     StorageScan = SSCAN_OFF;
991:     CycleControl = CYCLE_OFF;
992:     CheckControl = CHECK_STOP;
993:     DiskWrInhibit = false;
994:     AsteriskInsert = true;
995:     InhibitPrintOut = false;
996:     BitSwitches = BCD(0);
997:
998:     // Build the various displayable components of the CPU
999:
1000:    IRing = new TRingCounter(13);
1001:    assert(F1415L -> Light_I_OP != 0);
1002:    IRing -> lamps[0] = F1415L -> Light_I_OP;
1003:    IRing -> lamps[1] = F1415L -> Light_I_1;
1004:    IRing -> lamps[2] = F1415L -> Light_I_2;
1005:    IRing -> lamps[3] = F1415L -> Light_I_3;
1006:    IRing -> lamps[4] = F1415L -> Light_I_4;
1007:    IRing -> lamps[5] = F1415L -> Light_I_5;
1008:    IRing -> lamps[6] = F1415L -> Light_I_6;
1009:    IRing -> lamps[7] = F1415L -> Light_I_7;
1010:    IRing -> lamps[8] = F1415L -> Light_I_8;
1011:    IRing -> lamps[9] = F1415L -> Light_I_9;
1012:    IRing -> lamps[10] = F1415L -> Light_I_10;
1013:    IRing -> lamps[11] = F1415L -> Light_I_11;
1014:    IRing -> lamps[12] = F1415L -> Light_I_12;
1015:
1016:    ARing = new TRingCounter(6);
1017:    ARing -> lamps[0] = F1415L -> Light_A_1;
1018:    ARing -> lamps[1] = F1415L -> Light_A_2;
1019:    ARing -> lamps[2] = F1415L -> Light_A_3;
1020:    ARing -> lamps[3] = F1415L -> Light_A_4;
1021:    ARing -> lamps[4] = F1415L -> Light_A_5;
1022:    ARing -> lamps[5] = F1415L -> Light_A_6;
1023:
1024:    ClockRing = new TRingCounter(10);
1025:    ClockRing -> lamps[0] = F1415L -> Light_Clk_A;
1026:    ClockRing -> lamps[1] = F1415L -> Light_Clk_B;
1027:    ClockRing -> lamps[2] = F1415L -> Light_Clk_C;
1028:    ClockRing -> lamps[3] = F1415L -> Light_Clk_D;
1029:    ClockRing -> lamps[4] = F1415L -> Light_Clk_E;
1030:    ClockRing -> lamps[5] = F1415L -> Light_Clk_F;
1031:    ClockRing -> lamps[6] = F1415L -> Light_Clk_G;
1032:    ClockRing -> lamps[7] = F1415L -> Light_Clk_H;
1033:    ClockRing -> lamps[8] = F1415L -> Light_Clk_J;
1034:    ClockRing -> lamps[9] = F1415L -> Light_Clk_K;
1035:
1036:    ScanRing = new TRingCounter(4);
1037:    ScanRing -> lamps[0] = F1415L -> Light_Scan_N;
1038:    ScanRing -> lamps[1] = F1415L -> Light_Scan_1;
1039:    ScanRing -> lamps[2] = F1415L -> Light_Scan_2;
1040:    ScanRing -> lamps[3] = F1415L -> Light_Scan_3;
1041:
1042:    SubScanRing = new TRingCounter(5);
1043:    // NOTE: State 0 is "OFF" - no flip flops set
1044:    SubScanRing -> lamps[1] = F1415L -> Light_Sub_Scan_U;
1045:    SubScanRing -> lamps[2] = F1415L -> Light_Sub_Scan_B;
1046:    SubScanRing -> lamps[3] = F1415L -> Light_Sub_Scan_E;
1047:    SubScanRing -> lamps[4] = F1415L -> Light_Sub_Scan_MQ;
1048:
```

```
1049:     CycleRing = new TRingCounter(8);
1050:     CycleRing -> lamps[0] = F1415L -> Light_Cycle_A;
1051:     CycleRing -> lamps[1] = F1415L -> Light_Cycle_B;
1052:     CycleRing -> lamps[2] = F1415L -> Light_Cycle_C;
1053:     CycleRing -> lamps[3] = F1415L -> Light_Cycle_D;
1054:     CycleRing -> lamps[4] = F1415L -> Light_Cycle_E;
1055:     CycleRing -> lamps[5] = F1415L -> Light_Cycle_F;
1056:     CycleRing -> lamps[6] = F1415L -> Light_Cycle_I;
1057:     CycleRing -> lamps[7] = F1415L -> Light_Cycle_X;
1058:
1059: // The Cycle Ring also displays on the CE panel - special case
1060:
1061:     CycleRing -> lampsCE = new TLabel*[8];
1062:     CycleRing -> lampsCE[0] = F1415L -> Light_CE_Cyc_A;
1063:     CycleRing -> lampsCE[1] = F1415L -> Light_CE_Cyc_B;
1064:     CycleRing -> lampsCE[2] = F1415L -> Light_CE_Cyc_C;
1065:     CycleRing -> lampsCE[3] = F1415L -> Light_CE_Cyc_D;
1066:     CycleRing -> lampsCE[4] = F1415L -> Light_CE_Cyc_E;
1067:     CycleRing -> lampsCE[5] = F1415L -> Light_CE_Cyc_F;
1068:     CycleRing -> lampsCE[6] = F1415L -> Light_CE_Cyc_I;
1069:     CycleRing -> lampsCE[7] = F1415L -> Light_CE_Cyc_X;
1070:
1071:
1072: // Build the various latches. Most of these are not
1073: // reset during Program Reset, but some are.
1074:
1075: StopLatch = true;
1076: StopKeyLatch = false;
1077: DisplayModeLatch = false;
1078: StorageWrapLatch = false;
1079: ProcessRoutineLatch = false;
1080: BranchLatch = true;
1081: BranchToTollatch = true;
1082: LastInstructionReadout = false;
1083: IRingControl = true;
1084: IOChannelSelect = false;
1085: IndexLatches = 0;
1086:
1087: CarryIn = new TDisplayLatch(F1415L -> Light_Carry_In, false);
1088: CarryOut = new TDisplayLatch(F1415L -> Light_Carry_Out, false);
1089: AComplement = new TDisplayLatch(F1415L -> Light_A_Complement, false);
1090: BComplement = new TDisplayLatch(F1415L -> Light_B_Complement, false);
1091:
1092: CompareBGTA = new TDisplayLatch(F1415L -> Light_B_GT_A, false);
1093: CompareBEQA = new TDisplayLatch(F1415L -> Light_B_EQ_A, false);
1094: CompareBLTA = new TDisplayLatch(F1415L -> Light_B_LT_A, false);
1095: Overflow = new TDisplayLatch(F1415L -> Light_Overflow, false);
1096: DivideOverflow = new TDisplayLatch(F1415L -> Light_Divide_Overflow, false);
1097: ZeroBalance = new TDisplayLatch(F1415L -> Light_Zero_Balance, false);
1098:
1099: // Build the various check latches. Program Reset does reset these
1100:
1101: AChannelCheck = new TDisplayLatch(F1415L -> Light_Check_AChannel);
1102: BChannelCheck = new TDisplayLatch(F1415L -> Light_Check_BChannel);
1103: AssemblyChannelCheck = new TDisplayLatch(F1415L ->
    Light_Check_AssemblyChannel);
1104: AddressChannelCheck = new TDisplayLatch(F1415L ->
    Light_Check_AddressChannel);
1105: AddressExitCheck = new TDisplayLatch(F1415L -> Light_Check_AddressExit);
1106: ARegisterSetCheck = new TDisplayLatch(F1415L -> Light_Check_ARegisterSet);
1107: BRegisterSetCheck = new TDisplayLatch(F1415L -> Light_Check_BRegisterSet);
1108: OpRegisterSetCheck = new TDisplayLatch(F1415L -> Light_Check_OpRegisterSet);
1109: OpModifierSetCheck = new TDisplayLatch(F1415L -> Light_Check_OpModifierSet);
1110: ACharacterSelectCheck = new TDisplayLatch(F1415L ->
    Light_Check_ACharacterSelect);
1111: BCharacterSelectCheck = new TDisplayLatch(F1415L ->
```

```
    Light_Check_BCharacterSelect);

1112:     IOInterlockCheck = new TDisplayLatch(F1415L -> Light_Check_IOInterlock);
1113:     AddressCheck = new TDisplayLatch(F1415L -> Light_Check_AddressCheck);
1114:     RBCInterlockCheck = new TDisplayLatch(F1415L -> Light_Check_RBCInterlock);
1115:     InstructionCheck = new TDisplayLatch(F1415L ->
1116:                                         Light_Check_InstructionCheck);

1117:     // Build the Data Registers
1118:
1119:
1120:     A_Reg = new TRegister();
1121:
1122:     tmper = F1415L -> Light_CE_A_ER;
1123:     tmpl = new TLabel*[8];
1124:     tmpl[0] = F1415L -> Light_CE_A_1;
1125:     tmpl[1] = F1415L -> Light_CE_A_2;
1126:     tmpl[2] = F1415L -> Light_CE_A_4;
1127:     tmpl[3] = F1415L -> Light_CE_A_8;
1128:     tmpl[4] = F1415L -> Light_CE_A_A;
1129:     tmpl[5] = F1415L -> Light_CE_A_B;
1130:     tmpl[6] = F1415L -> Light_CE_A_C;
1131:     tmpl[7] = F1415L -> Light_CE_A_WM;
1132:     A_Reg -> SetDisplay(tmper,tmpl);
1133:
1134:     B_Reg = new TRegister();
1135:
1136:     tmper = F1415L -> Light_CE_B_ER;
1137:     tmpl = new TLabel*[8];
1138:     tmpl[0] = F1415L -> Light_CE_B_1;
1139:     tmpl[1] = F1415L -> Light_CE_B_2;
1140:     tmpl[2] = F1415L -> Light_CE_B_4;
1141:     tmpl[3] = F1415L -> Light_CE_B_8;
1142:     tmpl[4] = F1415L -> Light_CE_B_A;
1143:     tmpl[5] = F1415L -> Light_CE_B_B;
1144:     tmpl[6] = F1415L -> Light_CE_B_C;
1145:     tmpl[7] = F1415L -> Light_CE_B_WM;
1146:     B_Reg -> SetDisplay(tmper,tmpl);
1147:
1148:     Op_Reg = new TRegister();
1149:
1150:     DEBUG("B_Reg is at %x",B_Reg)
1151:     DEBUG("OP_Reg is at %x",Op_Reg)
1152:
1153:     tmpl = new TLabel*[8];
1154:     tmpl[0] = F1415L -> Light_CE_OP_1;
1155:     tmpl[1] = F1415L -> Light_CE_OP_2;
1156:     tmpl[2] = F1415L -> Light_CE_OP_4;
1157:     tmpl[3] = F1415L -> Light_CE_OP_8;
1158:     tmpl[4] = F1415L -> Light_CE_OP_A;
1159:     tmpl[5] = F1415L -> Light_CE_OP_B;
1160:     tmpl[6] = F1415L -> Light_CE_OP_C;
1161:     tmpl[7] = 0;
1162:     Op_Reg -> SetDisplay(0,tmpl);
1163:
1164:     Op_Mod_Reg = new TRegister();
1165:
1166:     tmpl = new TLabel*[8];
1167:     tmpl[0] = F1415L -> Light_CE_Mod_1;
1168:     tmpl[1] = F1415L -> Light_CE_Mod_2;
1169:     tmpl[2] = F1415L -> Light_CE_Mod_4;
1170:     tmpl[3] = F1415L -> Light_CE_Mod_8;
1171:     tmpl[4] = F1415L -> Light_CE_Mod_A;
1172:     tmpl[5] = F1415L -> Light_CE_Mod_B;
1173:     tmpl[6] = F1415L -> Light_CE_Mod_C;
1174:     tmpl[7] = 0;
1175:     Op_Mod_Reg -> SetDisplay(0,tmpl);
```

```
1176:  
1177: // Build the Address Registers  
1178:  
1179: STAR = new TAddressRegister();  
1180: A_AR = new TAddressRegister();  
1181: B_AR = new TAddressRegister();  
1182: C_AR = new TAddressRegister();  
1183: D_AR = new TAddressRegister();  
1184: E_AR = new TAddressRegister();  
1185: F_AR = new TAddressRegister();  
1186: I_AR = new TAddressRegister();  
1187:  
1188: // Build the channels  
1189:  
1190: Channel[CHANNEL1] = new T1410Channel(  
1191:     F1415L -> Light_Ch1_Interlock,  
1192:     F1415L -> Light_Ch1_RBCInterlock,  
1193:     F1415L -> Light_Ch1_Read,  
1194:     F1415L -> Light_Ch1_Write,  
1195:     F1415L -> Light_Ch1_Overlap,  
1196:     F1415L -> Light_Ch1_NoOverlap,  
1197:     F1415L -> Light_Ch1_NotReady,  
1198:     F1415L -> Light_Ch1_Busy,  
1199:     F1415L -> Light_Ch1_DataCheck,  
1200:     F1415L -> Light_Ch1_Condition,  
1201:     F1415L -> Light_Ch1_WLRecord,  
1202:     F1415L -> Light_Ch1_NoTransfer  
1203: );  
1204:  
1205: Channel[CHANNEL2] = new T1410Channel(  
1206:     F1415L -> Light_Ch2_Interlock,  
1207:     F1415L -> Light_Ch2_RBCInterlock,  
1208:     F1415L -> Light_Ch2_Read,  
1209:     F1415L -> Light_Ch2_Write,  
1210:     F1415L -> Light_Ch2_Overlap,  
1211:     F1415L -> Light_Ch2_NoOverlap,  
1212:     F1415L -> Light_Ch2_NotReady,  
1213:     F1415L -> Light_Ch2_Busy,  
1214:     F1415L -> Light_Ch2_DataCheck,  
1215:     F1415L -> Light_Ch2_Condition,  
1216:     F1415L -> Light_Ch2_WLRecord,  
1217:     F1415L -> Light_Ch2_NoTransfer  
1218: );  
1219:  
1220: // Build the A Channel and Assembly Channel  
1221:  
1222: AChannel = new TACchannel();  
1223: AssemblyChannel = new TAssemblyChannel();  
1224:  
1225: // Some latches are set after power on...  
1226:  
1227: CompareBLTA -> Set();  
1228: I_AR -> Set(1);  
1229: AdderResult = 0;  
1230:  
1231: // Initialize core-load  
1232:  
1233: for(i=0; i < strlen(core_load_chars); ++i) {  
1234:     core[i] = BCD::BCDConvert(core_load_chars[i]);  
1235:     if(core_load_wm[i]) {  
1236:         core[i].SetWM();  
1237:     }  
1238:     core[i].SetOddParity();  
1239: }  
1240:  
1241: // Finally, set up the indicators
```

```
1242:  
1243:     OffNormal = new TDisplayIndicator(F1415L -> Light_Off_Normal,  
1244:         &(CPU -> IndicatorOffNormal));  
1245:  
1246: }  
1247:  
1248:  
1249: // CPU Object display - run thru the display list  
1250:  
1251: void T1410CPU::Display()  
1252: {  
1253:     TDisplayObject *l;  
1254:  
1255:     for(l = DisplayList; l != 0; l = l -> NextDisplay) {  
1256:         l -> Display();  
1257:     }  
1258:  
1259: }  
1260:  
1261: // Off Normal Indicator routine  
1262:  
1263: bool T1410CPU::IndicatorOffNormal()  
1264: {  
1265:     return(CPU -> InhibitPrintOut ||  
1266:            !(CPU -> AsteriskInsert) ||  
1267:            CPU -> CycleControl != CPU -> CYCLE_OFF ||  
1268:            CPU -> CheckControl != CPU -> CHECK_STOP ||  
1269:            (CPU -> Mode == CPU -> MODE_CE &&  
1270:             CPU -> StorageScan != CPU -> SSCAN_OFF) ||  
1271:            CPU -> AddressEntry != CPU -> ADDR_ENTRY_I );  
1272: }  
1273:  
1274: // Cycle routine. Does typical kinds of CPU cycles  
1275:  
1276: void T1410CPU::Cycle()  
1277: {  
1278:     switch(CycleRing -> State()) {  
1279:  
1280:     case CYCLE_I:  
1281:  
1282:         *A_Reg = *B_Reg;  
1283:         AssemblyChannel -> GateAChannelToAssembly(  
1284:             AChannel -> Select(AChannel -> A_Channel_A));  
1285:         break;  
1286:  
1287:     case CYCLE_X:  
1288:         // This one is tricky, because it depends on opcode type.  
1289:         // So this is handled in the indexing routines  
1290:  
1291:         break;  
1292:  
1293:     default:  
1294:         break;  
1295:     }  
1296: }  
1297:  
1298: // Storage routine to read out (access) core storage.  
1299: // (A real core storage unit would also have to store it back)  
1300: // Address is in the STAR (aka MAR)  
1301:  
1302: void T1410CPU::Readout()  
1303: {  
1304:     long i;  
1305:  
1306:     // Get the contents of STAR: The Memory Address Register  
1307:
```

```
1308:     i = STAR -> Gate();           // Hee hee ;-) A Punny
1309:     StorageWrapLatch = false;
1310:
1311:     // This is really a debugging statement, but it's useful
1312:
1313:     if(i < 0) {
1314:         AddressCheck ->
1315:             SetStop("Address Check: STAR value not valid");
1316:         AddressExitCheck ->
1317:             SetStop("Address Exit Check: STAR value not valid");
1318:         B_Reg -> Set(0);           // Force a B Channel Check
1319:         return;
1320:     }
1321:
1322:     if(i > STORAGE) {
1323:         AddressCheck ->
1324:             SetStop("Address Check: STAR value > storage size");
1325:         B_Reg -> Set(0);
1326:         return;
1327:     }
1328:
1329:     // Check for a storage wrap condition. In most instances, it causes
1330:     // the CPU to stop with an Address Check, but there are exceptions.
1331:
1332:     if(i == STORAGE) {
1333:
1334:         // Valid wrap condition. Set address to 0, and set wrap condition.
1335:
1336:         i = 0;
1337:         StorageWrapLatch = true;
1338:     }
1339:
1340:     B_Reg -> Set(core[i]);
1341: }
1342:
1343: // Storage routine to store what is in the B Data Register
1344:
1345: void T1410CPU::Store()
1346: {
1347:     long i;
1348:
1349:     // Get the contents of STAR: The Memory Address Register
1350:
1351:     i = STAR -> Gate();
1352:
1353:     // This is really a debugging statement, but it's useful
1354:
1355:     if(i < 0) {
1356:         AddressCheck ->
1357:             SetStop("Address Check: STAR value not valid during store");
1358:         AddressExitCheck ->
1359:             SetStop("Address Exit Check: STAR value not valid during store");
1360:         return;
1361:     }
1362:
1363:     if(i > STORAGE) {
1364:         AddressCheck ->
1365:             SetStop("Address Check: Star value > size of storage during store");
1366:         return;
1367:     }
1368:
1369:     // Check for a storage wrap condition.
1370:
1371:     if(i == STORAGE) {
1372:
1373:         // Valid wrap condition. Set address to 0
```

```
1374:
1375:         i = 0;
1376:         StorageWrapLatch = true;
1377:     }
1378:
1379:
1380:
1381:     if(!B_Reg -> Get().CheckParity()) {
1382:         BChannelCheck ->
1383:             SetStop("B Channel Check: Invalid parity");
1384:     }
1385:
1386:     core[i] = B_Reg -> Get();
1387: }
1388:
1389: // Storage routine to store data (Assembly Channel is implied here)
1390:
1391: void T1410CPU::Store(BCD bcd)
1392: {
1393:     B_Reg -> Set(bcd);
1394:
1395:     if(!B_Reg -> Get().CheckParity()) {
1396:         AssemblyChannelCheck ->
1397:             SetStop("Assembly Channel Check: B Channel invalid during store");
1398:     }
1399:
1400:     Store();
1401: }
1402:
1403: // Set Storage Scan Mode
1404:
1405: void T1410CPU::SetScan(char i)
1406: {
1407:     ScanRing -> Set(i);
1408: }
1409:
1410: // Apply storage scan value to STAR and return result.
1411: // Essentially this gates the ScanControl Ring to the Address Modification
1412: // circuitry
1413:
1414: long T1410CPU::STARScan()
1415: {
1416:     return(STAR -> Gate() + scan_mod[ScanRing -> State()]);
1417: }
1418:
1419: // Apply +1 / 0 / -1 modification value to STAR and return result.
1420: // This amounts to direct use of the address modification circuitry when
1421: // not using SCAN CTRL (e.g. when doing Instruction readout)
1422:
1423: long T1410CPU::STARMod(int mod)
1424: {
1425:     return(STAR -> Gate() + mod);
1426: }
1427:
1428: // Method to load core from a file
1429:
1430: void T1410CPU::LoadCore(char *filename)
1431: {
1432:     FILE *fd;
1433:     long coresize,loc;
1434:     char file_coresize[6];
1435:     int file_core[STORAGE];
1436:
1437:     // First open the file.
1438:
1439:     if((fd = fopen(filename,"rb")) == NULL) {
```

```
1440:         Application -> MessageBox("Unable to open file to load core.",  
1441:             "Load Core Error",MB_OK);  
1442:         return;  
1443:     }  
1444:  
1445:     // Next, read in the (5 digit) core size  
1446:  
1447:     file_coresize[sizeof(file_coresize)-1] = '\0';  
1448:     if(fread(&file_coresize,1,5,fd) != 5) {  
1449:         Application -> MessageBox("Error reading dump core size from file.",  
1450:             "Error Reading Dump",MB_OK);  
1451:         fclose(fd);  
1452:         return;  
1453:     }  
1454:  
1455:     // Convert from decimal to long, and validate.  
1456:  
1457:     coresize = atol(file_coresize);  
1458:     if(coresize < 0) {  
1459:         Application -> MessageBox("Dump contained negative value for core  
size.",  
1460:             "Negative Core Size in File",MB_OK);  
1461:         fclose(fd);  
1462:         return;  
1463:     }  
1464:  
1465:     if(coresize > STORAGE) {  
1466:         Application -> MessageBox("Dump contained core size greater than  
simulator's.",  
1467:             "Large Core Size in File",MB_OK);  
1468:         fclose(fd);  
1469:         return;  
1470:     }  
1471:  
1472:     // Read in the data from the dump file.  
1473:  
1474:     if(fread(&file_core,sizeof(int),coresize,fd) != coresize) {  
1475:         Application -> MessageBox("Error loading core data from file",  
1476:             "Error Loading Core",MB_OK);  
1477:         fclose(fd);  
1478:     }  
1479:  
1480:     // Finally, copy the data to core.  
1481:  
1482:     for(loc=0; loc < coresize; ++loc) {  
1483:         core[loc].Set(file_core[loc]);  
1484:     }  
1485:  
1486:     fclose(fd);  
1487:     Application -> MessageBox("Core Loaded!","Core Loaded",MB_OK);  
1488: }  
1489:  
1490: void T1410CPU::DumpCore(char *filename)  
1491: {  
1492:     FILE *fd;  
1493:     long loc;  
1494:     int file_core[STORAGE];  
1495:     char file_coresize[6];  
1496:  
1497:     // First, prepare a file.  
1498:  
1499:     if((fd = fopen(filename,"wb")) == NULL) {  
1500:         Application -> MessageBox("Unable to open file to dump core.",  
1501:             "Dump Core Error",MB_OK);  
1502:     }  
1503:
```

```
1504:     // Then, copy core to an integer array
1505:
1506:     for(loc=0; loc < STORAGE; ++loc){
1507:         file_core[loc] = core[loc].ToInt();
1508:     }
1509:
1510:     // Write out the core size to the file
1511:
1512:     if(fprintf(fd,"%05d",STORAGE) != 5) {
1513:         Application -> MessageBox("Error writing out core size.",
1514:             "Core Size Write Error",MB_OK);
1515:         fclose(fd);
1516:         return;
1517:     }
1518:
1519:     // Then write out the integer array
1520:
1521:     if(fwrite(&file_core,sizeof(int),STORAGE,fd) != STORAGE) {
1522:         Application -> MessageBox("Error writing out core file.",
1523:             "File Write Error",MB_OK);
1524:         fclose(fd);
1525:     }
1526:
1527:     Application -> MessageBox("Core Dumped!","Core Dumped",MB_OK);
1528:     fclose(fd);
1529: }
1530:
1531: // The 1410 Adder. It works by translating the numeric part into
1532: // QuiBinary code. The real 1410 used logic gates to add, we use
1533: // a table.
1534:
1535: // Here is the code
1536:
1537: #define ADDER_BINARY_0 0
1538: #define ADDER_BINARY_1 1
1539: #define ADDER_QUINARY_0 0
1540: #define ADDER_QUINARY_1 1
1541: #define ADDER_QUINARY_2 2
1542: #define ADDER_QUINARY_3 3
1543: #define ADDER_QUINARY_4 4
1544:
1545: // This table translates the numeric part of a BCD character (8421) to
1546: // obtain the quinary part
1547:
1548: static int num_to_true_quinary[] = {
1549:     ADDER_QUINARY_0,                      // 0 (no bits)
1550:     ADDER_QUINARY_0,                      // 1
1551:     ADDER_QUINARY_2,                      // 2
1552:     ADDER_QUINARY_2,                      // 3 (2 + 1)
1553:     ADDER_QUINARY_4,                      // 4
1554:     ADDER_QUINARY_4,                      // 5 (4 + 1)
1555:     ADDER_QUINARY_6,                      // 6
1556:     ADDER_QUINARY_6,                      // 7 (6 + 1)
1557:     ADDER_QUINARY_8,                      // 8
1558:     ADDER_QUINARY_8,                      // 9 (8 + 1)
1559:     ADDER_QUINARY_0,                      // 0 (8 + 2)
1560:     ADDER_QUINARY_2,                      // 3 (2 + 1) (8 is ignored)
1561:     ADDER_QUINARY_4,                      // 4 (8 is ignored)
1562:     ADDER_QUINARY_4,                      // 5 (4+ 1) (8 is ignored)
1563:     ADDER_QUINARY_6,                      // 6 (8 is ingored)
1564:     ADDER_QUINARY_6,                      // 7 (6 + 1) (8 is ignored)
1565: };
1566:
1567: // This table does the same thing, but generated the complement value,
1568: // used for subtraction and negative numbers. Note that these entries
1569: // are 8 - (the entry from the above table). But we are generated *indexes*
```

```
1570: // here, not actual values, so we couldn't just subtract!
1571:
1572: static int num_to_complement_quinary[] = {
1573:     ADDER_QUINARY_8, ADDER_QUINARY_8, ADDER_QUINARY_6, ADDER_QUINARY_6,
1574:     ADDER_QUINARY_4, ADDER_QUINARY_4, ADDER_QUINARY_2, ADDER_QUINARY_2,
1575:     ADDER_QUINARY_0, ADDER_QUINARY_0, ADDER_QUINARY_8, ADDER_QUINARY_6,
1576:     ADDER_QUINARY_4, ADDER_QUINARY_4, ADDER_QUINARY_2, ADDER_QUINARY_2
1577: };
1578:
1579: // Adder Quinary Matrix. Combines the two quinary values, returning
1580: // a quinary result, and a carry value. Indexed as [B][A], but the
1581: // table is symmetric (addition is complementary, even in quinary matrices!)
1582:
1583: struct adder_matrix {
1584:     int quinary;
1585:     bool carry;
1586: } adder_matrix[5][5] = {
1587:     { { ADDER_QUINARY_0, false }, { ADDER_QUINARY_2, false }, // 0-0, 0-2
1588:       { ADDER_QUINARY_4, false }, { ADDER_QUINARY_6, false }, // 0-4, 0-6
1589:       { ADDER_QUINARY_8, false } },
1590:
1591:     { { ADDER_QUINARY_2, false }, { ADDER_QUINARY_4, false }, // 2-0, 2-2
1592:       { ADDER_QUINARY_6, false }, { ADDER_QUINARY_8, false }, // 2-4, 2-6
1593:       { ADDER_QUINARY_0, true } },
1594:
1595:     { { ADDER_QUINARY_4, false }, { ADDER_QUINARY_6, false }, // 4-0, 4-2
1596:       { ADDER_QUINARY_8, false }, { ADDER_QUINARY_0, true }, // 4-4, 4-6
1597:       { ADDER_QUINARY_2, true } },
1598:
1599:     { { ADDER_QUINARY_6, false }, { ADDER_QUINARY_8, false }, // 6-0, 6-2
1600:       { ADDER_QUINARY_0, true }, { ADDER_QUINARY_2, true }, // 6-4, 6-6
1601:       { ADDER_QUINARY_4, true } },
1602:
1603:     { { ADDER_QUINARY_8, false }, { ADDER_QUINARY_0, true }, // 8-0, 8-2
1604:       { ADDER_QUINARY_2, true }, { ADDER_QUINARY_4, true }, // 8-4, 8-6
1605:       { ADDER_QUINARY_6, true } }
1606: };
1607:
1608: // Adder result table. Indexed via [quinary value][bshift value]
1609:
1610: int adder_result_table[5][4] = {
1611:     { 0, 1, 2, 3 },
1612:     { 2, 3, 4, 5 },
1613:     { 4, 5, 6, 7 },
1614:     { 6, 7, 8, 9 },
1615:     { 8, 9, 0, 1 }
1616: };
1617:
1618: // Adder.
1619:
1620: // Note: eventually we can make this more efficient by just generating
1621: // a 16x16 table for all of the possibilities during the CPU constructor!
1622:
1623: BCD T1410CPU::Adder(BCD A,int Complement_A,BCD B,int Complement_B)
1624: {
1625:     int bshift;
1626:     struct adder_matrix *qmatrix;
1627:     int bin;
1628:
1629:     int bcd_a,bcd_b;
1630:     int quinary_a,quinary_b;
1631:     int binary_a,binary_b;
1632:
1633:     bcd_a = A.ToInt();
1634:     bcd_b = B.ToInt();
1635:
```

```
1636: // Based on the complement requests, translate BCD to QuiBinary.
1637:
1638: if(Complement_A) {
1639:     quinary_a = num_to_complement_quinary[bcd_a & 0x0f];
1640:     binary_a = 1 - (bcd_a & 1);
1641: }
1642: else {
1643:     quinary_a = num_to_true_quinary[bcd_a & 0x0f];
1644:     binary_a = bcd_a & 1;
1645: }
1646:
1647: if(Complement_B) {
1648:     quinary_b = num_to_complement_quinary[bcd_b & 0x0f];
1649:     binary_b = 1 - (bcd_b & 1);
1650: }
1651: else {
1652:     quinary_b = num_to_true_quinary[bcd_b & 0x0f];
1653:     binary_b = bcd_b & 1;
1654: }
1655:
1656: // Handle the binary parts by adding them together along with the
1657: // CarryIn latch. This results in a number from 0 to 3.
1658:
1659: bshift = binary_a + binary_b + (CarryIn -> State());
1660:
1661: // Add the Quinary parts together according to the matrix
1662:
1663: qmatrix = &adder_matrix[quinary_a][quinary_b];
1664:
1665: // Set the carry out state...
1666:
1667: CarryOut -> Set(qmatrix -> carry ||
1668:                     (qmatrix -> quinary == ADDER_QUINARY_8 && bshift > 1));
1669:
1670: // Calculate the binary result
1671:
1672: bin = adder_result_table[qmatrix -> quinary][bshift];
1673:
1674: // Finally, translate the binary result to BCD. Since we *know* that
1675: // the bin value is range 0-9, we just cheat, and use the ascii to bcd
1676: // table
1677:
1678: AdderResult = BCD::BCDConvert(bin + '0');
1679: return(AdderResult);
1680: }
```



```
67:     // Note that we don't do the address modification cycle yet, as we
68:     // have to store the result back where BAR points!
69:
70:     // The following looks complicated on the flow chart in the
71:     // IBM CE Instructional materials 1411 Processing Unit Instructions
72:     // on page 11. But, the end result is that if you have an EVEN number
73:     // of: -A, -B and SUBTRACT you do a TRUE ADD cycle, and ODD number and
74:     // you do a COMPLEMENT ADD cycle
75:
76:     if(ScanRing -> State() == SCAN_1) {           // Is 1st scan ring on?
77:         if(SubScanRing -> State() == SUB_SCAN_U) { // Units latch?
78:             BComplement -> Reset();                // Set True Add B latch
79:             i =
80:                 (A_Reg -> Get().IsMinus()) +
81:                 (B_Reg -> Get().IsMinus()) +
82:                 (op_bin == OP_SUBTRACT);
83:             if(i & 1) {
84:                 AComplement -> Set();                // Odd #: complement add
85:                 CarryIn -> Set();
86:             }
87:             else {
88:                 AComplement -> Reset();              // Even #: true add
89:                 CarryIn -> Reset();
90:             }
91:
92:             adder_a = A_Reg -> Get();
93:
94:         } // End, units
95:     else {
96:         if(SubScanRing -> State() == SUB_SCAN_E) { // Extension latch?
97:             adder_a = BCD_0;                         // Yes, gate '0'
98:                                         // (if a compl then 9)
99:
100:    }
101:    else {
102:        adder_a = A_Reg -> Get();                // No, use A data
103:    }
104: } // End, SCAN 1
105:
106: else {
107:     assert(ScanRing -> State() == SCAN_3);      // Else must be 3
108:     if(SubScanRing -> State() == SUB_SCAN_U) {    // Units?
109:         b_temp = B_Reg -> Get();                  // Yes. Invert sign
110:         b_temp = (b_temp & ~BIT_ZONE) |
111:             (b_temp.IsMinus() ? (BITA | BITB) : BITB);
112:         b_temp.SetOddParity();
113:         B_Reg -> Set(b_temp);
114:         AComplement -> Reset();                  // Reset compl add
115:     }
116:     adder_a = BCD_0;
117: }
118:
119: // Run the right stuff thru the adder, and then thru the Assembly Channel
120:
121: Adder(addr_a,AComplement -> State(),
122:       B_Reg -> Get(),BComplement -> State());
123:
124: if(ZeroBalance -> State() &&
125:     (AdderResult & BIT_NUM) != (BCD_0 & ~BITC)) {
126:     ZeroBalance -> Reset();
127: }
128:
129: AssemblyChannel -> Select(
130:     AssemblyChannel -> AsmChannelWMB,
131:     AssemblyChannel -> AsmChannelZonesB,
132:     false,
```

```
133:     AssemblyChannel -> AsmChannelSignNone,
134:     AssemblyChannel -> AsmChannelNumAdder
135:   );
136:
137:   if(B_Reg -> Get().TestWM()) { // B Channel WM?
138:
139:     if(ScanRing -> State() == SCAN_3) { // 3rd scan latch?
140:       Store(AssemblyChannel -> Select()); // Store the last char
141:       Cycle(); // Modify B Addr -1
142:       IRingControl = true; // Done!
143:       return;
144:     }
145:
146:     if(AComplement -> State()) { // Complement add?
147:
148:       if(CarryOut -> State()) { // Carry too?
149:         Store(AssemblyChannel -> Select()); // Store the last char
150:         Cycle(); // Modify B addr -1
151:         IRingControl = true; // Done! (no sign sw)
152:         return;
153:       }
154:
155:       // What I *think* the following does is to reverse the B sign
156:       // where D_AR points, because SUB_SCAN_U causes the CPU to use
157:       // the DAR for the storage cycle, which starts a recomplement
158:       // cycle wherein the entire B field is recomplemented. This
159:       // can only happen if we are doing a complement add,
160:       // with no carry out.
161:
162:       SubScanRing -> Set(SUB_SCAN_U); // Set units latch
163:       ScanRing -> Set(SCAN_3); // Set 3rd scan latch
164:       BComplement -> Set(); // Set B compl add
165:       CarryIn -> Set(); // Set carry
166:       Store(AssemblyChannel -> Select()); // Store the last char
167:       Cycle(); // Finish the B cycle
168:       cycle_type = CYCLE_B; // Re-enter at B cycle
169:       return;
170:     }
171:
172:   else { // True add
173:     if(CarryOut -> State()) { // Set overflow on carry
174:       Overflow -> Set();
175:     }
176:     Store(AssemblyChannel -> Select()); // Store the last char
177:     Cycle(); // Modify B addr -1
178:     IRingControl = true; // Done!
179:     return;
180:   }
181: } // End B Ch WM
182:
183: else { // No B Ch WM
184:   CarryIn -> Set(CarryOut -> State()); // Set carry
185:   Store(AssemblyChannel -> Select()); // Store result char
186:   Cycle(); // Modify B address -1
187:   if(ScanRing -> State() == SCAN_1) { // Still 1st scan?
188:     if(A_Reg -> Get().TestWM()) { // A Ch WM?
189:       SubScanRing -> Set(SUB_SCAN_E); // Yes -- set extension
190:       cycle_type = CYCLE_B; // No more A cycles!
191:       return; // continue on
192:     }
193:     else { // No A CH WM
194:       SubScanRing -> Set(SUB_SCAN_B); // Set to body
195:       cycle_type = CYCLE_A; // A cycle next
196:       return;
197:     }
198:   }
}
```

```
199:     else {
200:         assert(ScanRing -> State() == SCAN_3);           // Must be!
201:         SubScanRing -> Set(SUB_SCAN_E);                 // Still in extension
202:         cycle_type = CYCLE_B;                           // Which means no A cyc
203:         return;
204:     }
205: }
206: }
207:
208: void T1410CPU::InstructionZeroArith()
209: {
210:
211:     // Variable to maintain state between cycles
212:
213:     static int cycle_type;
214:     static int op_bin;
215:
216:     BCD a_temp,a_hold;
217:
218:     // If this is the first time in, set things up
219:
220:     if(LastInstructionReadout) {
221:         SubScanRing -> Set(SUB_SCAN_U);                  // Units
222:         ScanRing -> Set(SCAN_1);                      // Scan 1 (-1 mod)
223:
224:         AComplement -> Reset();
225:         BComplement -> Reset();
226:         CarryIn -> Reset();
227:         CarryOut -> Reset();
228:         ZeroBalance -> Set();                          // Assume 0 at start
229:
230:         cycle_type = CYCLE_A;                           // Set for 1st A cycle
231:         op_bin = Op_Reg -> Get().ToInt() & 0x3f;
232:         LastInstructionReadout = false;
233:     }
234:
235:     // Initial A cycle
236:
237:     if(cycle_type == CYCLE_A) {
238:         CycleRing -> Set(CYCLE_A);                   // Take an A Cycle
239:         *STAR = *A_AR;
240:         Readout();                                     // RO A field char.
241:         Cycle();                                       // Set AAR with mod
242:
243:         // Units or body latch Regen happens by itself in simulator
244:
245:         cycle_type = CYCLE_B;                         // Set for B cycle
246:         return;
247:     }
248:
249:     assert(cycle_type == CYCLE_B);                   // Has to be B cycle
250:
251:     CycleRing -> Set(CYCLE_B);                   // Take B Cycle
252:     if(SubScanRing -> State() == SUB_SCAN_U) {      // Read B units via DAR
253:         *STAR = *D_AR;
254:     }
255:     else {
256:         *STAR = *B_AR;
257:     }
258:     Readout();                                     // Ro B Field Char
259:
260:     // If we are in the body/extension, we just insert 0's on the
261:     // A channel. Otherwise, we normalize the A field sign
262:
263:     a_temp = A_Reg -> Get();                      // Save to restore ...
264:     a_hold = a_temp;
```



```
331:      }  
332: }
```

```
1: //-----
2: #include <vcl\vcl.h>
3: #pragma hdrstop
4:
5: #include "UI1410PWR.h"
6: //-----
7: #pragma resource "* .dfm"
8:
9: #include "UI1410DEBUG.h"
10:
11: // These methods define the operation of the 1410 power panel
12:
13: TFI1410PWR *FI1410PWR;
14: //-----
15: __fastcall TFI1410PWR::TFI1410PWR(TComponent* Owner)
16:     : TForm(Owner)
17: {
18:     Mode -> ItemIndex = CPU -> MODE_RUN;
19:     Height = 522;
20:     Width = 327;
21: }
22: //-----
23: void __fastcall TFI1410PWR::EmergencyOffClick(TObject *Sender)
24: {
25:     FI14101 -> Close();
26: }
27: //-----
28: void __fastcall TFI1410PWR::ComputerResetClick(TObject *Sender)
29: {
30:     TCpuObject *o;
31:
32:     DEBUG("Computer Reset",0)
33:     for(o = CPU -> ResetList; o != 0; o = o -> NextReset) {
34:         o -> OnComputerReset();
35:     }
36:
37:     // Handle any special case latches (which are reset in the above loop)
38:
39:     CPU -> CompareBLTA -> Set();
40:     CPU -> I_AR -> Set(1);
41:     CPU -> StopLatch = true;
42:
43:     StopClick(Sender);
44:
45:     CPU -> Display();
46:
47:     // Reset the console
48:
49:     FI1415IO -> ResetMatrix();
50:     FI1415IO -> NextLine();
51:     FI1415IO -> SetState(CONSOLE_IDLE);
52: }
53: //-----
54: void __fastcall TFI1410PWR::ModeChange(TObject *Sender)
55: {
56:     if(Mode -> ItemIndex >= 0) {
57:         CPU -> Mode = Mode -> ItemIndex;
58:     }
59:     CPU -> OffNormal -> Display();
60:     DEBUG("Mode Switch set to %d",CPU -> Mode)
61: }
62: //-----
63: void __fastcall TFI1410PWR::ProgramResetClick(TObject *Sender)
64: {
65:     TCpuObject *o;
66:
```

```
67:     DEBUG("Program Reset",0)
68:     for(o = CPU -> ResetList; o != 0; o = o -> NextReset) {
69:         o -> OnProgramReset();
70:     }
71:
72:     // Handle any special cases
73:
74:     CPU -> I_AR -> Set(1);
75:     CPU -> StopLatch = true;
76:
77:     StopClick(Sender);
78:
79:     CPU -> Display();
80: }
81: //-----
82:
83: void __fastcall TFI1410PWR::StartClick(TObject *Sender)
84: {
85:     switch(CPU -> Mode) {
86:
87:     case CPU -> MODE_ADDR:
88:         FI1415IO -> DoAddressEntry();
89:         break;
90:
91:     case CPU -> MODE_DISPLAY:
92:         FI1415IO -> DoDisplay(1);
93:         break;
94:
95:     default:
96:         break;
97:     }
98:
99: }
100: //-----
101: void __fastcall TFI1410PWR::StopClick(TObject *Sender)
102: {
103:     CPU -> StopKeyLatch = true;
104:     FI1415IO -> DoDisplay(4);
105: }
106: //-----
```

```
1: //-----
2: #ifndef UI1410CPUUTH
3: #define UI1410CPUUTH
4:
5: #include "ubcd.h"
6:
7: extern long ten_thousands[],thousands[],hundreds[],tens[];
8: extern long scan_mod[];
9: extern unsigned char sign_normalize_table[];
10: extern unsigned char sign_complement_table[];
11:
12: //
13: // Classes (types) used to implement the emulator, including the
14: // final class defining what is in the CPU, T1410CPU.
15:
16: //
17: // Abstract class designed to build lists of objects affected by Program
18: // Reset and Computer Reset
19: //
20:
21: class TCpuObject : public TObject {
22:
23: public:
24:     TCpuObject();                                // Constructor to init data
25:     virtual void OnComputerReset() = 0;           // Called during Computer Reset
26:     virtual void OnProgramReset() = 0;             // Called during Program Reset
27:
28: protected:
29:     bool DoesProgramReset;                      // true if this is reset by P.R. button
30:
31: public:
32:     TCpuObject *NextReset;
33: };
34:
35:
36: //
37: // A second abstract class of objects that not only react to the Resets,
38: // but also have entries on the display panel.
39: //
40:
41: class TDisplayObject : public TCpuObject {
42:
43: public:
44:     TDisplayObject();                            // Constructor to init data.
45:     virtual void Display() = 0;                  // Called to display this item
46:     virtual void LampTest(bool b) = 0;            // Called to start/end lamp test
47:
48: public:
49:     TDisplayObject *NextDisplay;
50: };
51:
52: □
53:
54: // Class TDisplayObjects are indicators: They just
55: // display other things. As a result, they need a pointer to
56: // a function returning bool in order to decide what to do.
57:
58: class TDisplayIndicator : public TDisplayObject {
59:
60: protected:
61:     TLabel *lamp;
62:     bool (_closure *display)();
63:
64: public:
65:
66:     // The constructor requires a pointer to a lamp and a pointer to
```

```
67: // a function that can calculate lamp state.
68:
69: // We use a closure so that we don't have to pass a pointer to the
70: // stuff the display function needs (an object pointer is automatically
71: // embedded in an __closure *)
72:
73: TDisplayIndicator(TLabel *l,bool (__closure *func)() ) {
74:     lamp = l;
75:     display = func;
76: }
77:
78: virtual void OnComputerReset() { ; }      // These have no state to reset
79: virtual void OnProgramReset() { ; }        // These have no state to reset
80:
81: void Display() {
82:     lamp -> Enabled = display();
83:     lamp -> Repaint();
84: }
85:
86: void LampTest(bool b) {
87:     lamp -> Enabled = (b ? true : display());
88:     lamp -> Repaint();
89: }
90: };
91:
92: □
93:
94: //
95: // Class of TDisplayObjects that are latches:
96: // that can be set, reset and their state read out. Some are
97: // reset by Program Reset (PR) some are not.
98: //
99:
100: class TDisplayLatch : public TDisplayObject {
101:
102: protected:
103:     bool state;                                // Latches can be set or reset
104:     bool doprogramreset;                      // Some are reset by PR, some are not.
105:     TLabel *lamp;                            // Pointer to display lamp.
106:
107: public:
108:     TDisplayLatch(TLabel *l);                // Constructor - Set up lamp
109:     TDisplayLatch(TLabel *l, bool progreset); // Same, but inhibit PR
110:
111:     virtual void OnComputerReset();          // Define Computer Reset behavior now
112:     virtual void OnProgramReset();           // Define Program Reset behavior now too
113:
114:     void Display();                         // Define display behavior
115:     void LampTest(bool b);                  // Define lamp test behavior.
116:
117:     // All you can really do with latches is set/reset/test them
118:
119:     inline void Reset() { state = false; }
120:     inline void Set() { state = true; }
121:     inline void Set(bool b) { state = b; }
122:     void SetStop();
123:     inline bool State() { return state; }
124: };
125:
126: □
127:
128: class TRingCounter : public TDisplayObject {
129: private:
130:     char state;                                // Override "state" variable !!
131:     char max;                                  // Max number of entries
132:     TLabel *lastlamp;                        // Last lamp to be displayed
```

```
133:     TLabel *lastlampCE;           // Last CE lamp to be displayed
134:
135: public:
136:     TLabel **lamps;             // Ptr to array of lamps
137:     TLabel **lampsCE;          // Ptr to array of CE lamps (or 0)
138:
139: public:
140:     TRingCounter(char n);       // Construct with # of entries
141:                               // Ring counters are always reset by PR
142:
143:     virtual __fastcall ~TRingCounter(); // Destructor for array of lamps
144:
145: // Functions inherited from abstract base classes now need definition
146:
147:     void OnComputerReset();
148:     void OnProgramReset();
149:     void Display();
150:     void LampTest(bool b);
151:
152: // The real meat of the Ring Counter class
153:
154:     inline void Reset() { state = 0; }
155:     inline char Set(char n) { return state = n; }
156:     inline char State() { return state; }
157:     char Next();
158: };
159:
160: □
161:
162: // Data Registers. All are stored as BCD, but we have special
163: // set routines to set or clear special parts for those registers
164: // that don't use all the bits (e.g. the Op register has no WM or
165: // C bits.
166:
167: // Also, a Register can have an optional pointer to an error latch.
168: // If so, then whenever the register is used, a parity check for ODD
169: // parity is made, and if invalid, the error latch is set. (This
170: // includes use during assignment).
171:
172: class TRegister : public TDisplayObject {
173:
174: private:
175:
176:     BCD value;
177:     TLabel *lampER;           // If set, there is an error lamp
178:     TLabel **lamps;           // If set, they point to lamps for WM C B A 8 4 2 1
179:                               // Any lamp not present MUST be set to 0
180:
181: public:
182:
183:     TRegister() { value = BITC; DoesProgramReset = true; lampER = 0; lamps = 0; }
184:     TRegister(bool b) { value = BITC; DoesProgramReset = b; lampER = 0; lamps =
185:     0; }
186:     TRegister(int i) { value = i; DoesProgramReset = true; lampER = 0; lamps = 0;
187:     }
188:     TRegister(int i, bool b) { value = i; DoesProgramReset = b; lampER = 0; lamps =
189:     0; }
190:
191:     inline void OnComputerReset() { Reset(); }
192:     inline void Reset() { value = BITC; }
193:
194:     inline void Set(BCD bcd) { value = bcd; }
195:     inline BCD Get() { return value; }
196:
197:     void operator=(TRegister &source);
```

```
196:     void Display();
197:     void LampTest(bool b);
198:
199:     // To set up a register to display, provide a pointer to an error
200:     // lamp (if any), and an array of pointers to data lamps. Data lamps
201:     // for any given bit (e.g. WM) may or may not exist. Order of lamps
202:     // is 1 2 4 8 A B C WM (0 thru 7)
203:
204:     void SetDisplay(TLabel *ler, TLabel **l) {
205:         lampER = ler;
206:         lamps = l;
207:     }
208:
209:     void OnProgramReset() {
210:         if(DoesProgramReset) {
211:             Reset();
212:         }
213:     }
214: };
215: □
216: // Address Registers. For efficiency, we keep both binary and
217: // the real 2-out-of-5 code representations. If either one is
218: // valid, and the other representation is requested, we convert
219: // on the fly (and mark that representation valid).
220:
221: // The Set() function effectively implements the Address Channel.
222:
223: class TAddressRegister : public TCpuObject {
224:
225: private:
226:
227:     long i_value;           // Integer equivalent of register
228:     bool i_valid;          // True if integer rep. is valid
229:     bool set[5];           // True if corresponding digit is set
230:     TWOOF5 digits[5];      // Original 2 out of 5 code representation
231:     bool d_valid;          // True if digit rep. is valid
232:
233: public:
234:
235:     void OnComputerReset() { Reset(); };
236:     void OnProgramReset() { };
237:
238:     TAddressRegister();    // Constructor / initialization
239:     bool IsValid();        // Returns true if all digits set
240:     long Gate();           // Returns integer value if valid, -1 if not.
241:     BCD GateBCD(int i);   // Returns a single digit
242:     void Set(TWOOF5 digit,int index); // Sets a digit
243:     void Set(long value); // Sets whole register from binary (address mod)
244:     void Reset();          // Resets the register to blanks
245:
246:     void operator=(TAddressRegister &source); // Assignment
247:
248: };
249:
250: □
251:
252: // This class defines the A Channel - basically, it defines what
253: // register is selected when the A channel is accessed.
254:
255: class TAChannel : public TDisplayObject {
256:
257: public:
258:
259:     enum AChannelSelect { A_Channel_None = 0, A_Channel_A = 1,
260:                           A_Channel_Mod = 2, A_Channel_E = 3, A_Channel_F = 4 };
261:
```

```
262: private:
263:
264:     enum AChannelSelect AChannelSelect;
265:     TLabel *lamps[4];
266:
267: public:
268:
269:     TAChannel();                                // Constructor
270:
271:     void OnComputerReset() {
272:         Reset();
273:     }
274:
275:     void OnProgramReset() {
276:         Reset();
277:     }
278:
279:     inline BCD Select(enum AChannelSelect sel); // Select input to A Channel
280:     inline BCD Select();                      // Use whatever was last selected
281:
282:     inline void Reset() { AChannelSelect = A_Channel_None; }
283:
284:     void Display();
285:     void LampTest(bool b);
286: };
287:
288: □
289:
290: // The Assembly Channel: The 1410 Mix-Master!
291:
292: class TAssemblyChannel : public TDisplayObject {
293:
294: public:
295:
296:     enum AsmChannelZonesSelect {
297:         AsmChannelZonesNone = 0, AsmChannelZonesB = 1, AsmChannelZonesA = 2
298:     };
299:
300:     enum AsmChannelWMSelect {
301:         AsmChannelWMNone = 0, AsmChannelWMB = 1, AsmChannelWMA = 2
302:     };
303:
304:     enum AsmChannelNumericSelect {
305:         AsmChannelNumNone = 0, AsmChannelNumB = 1, AsmChannelNumA = 2,
306:         AsmChannelNumAdder = 3
307:     };
308:
309:     enum AsmChannelSignSelect {
310:         AsmChannelSignNone, AsmChannelSignA = 1, AsmChannelSignB = 2,
311:         AsmChannelSignLatch = 3
312:     };
313:
314: private:
315:
316:     BCD value;                                // We remember value, for efficiency
317:     bool valid;                               // True if value is set. Reset each cycle!
318:
319:     enum AsmChannelZonesSelect AsmChannelZonesSelect;
320:     enum AsmChannelWMSelect AsmChannelWMSelect;
321:     enum AsmChannelNumericSelect AsmChannelNumericSelect;
322:     enum AsmChannelSignSelect AsmChannelSignSelect;
323:     bool AsmChannelInvertSign;
324:
325:     TLabel *AssmLamps[8];
326:     TLabel *AssmComplLamps[8];
327:     TLabel *AssmERLamp;
```

```

328:
329: public:
330:
331:     TAssemblyChannel();
332:
333:     void OnComputerReset() { Reset(); }
334:     void OnProgramReset() { Reset(); }
335:
336:     void Display();
337:     void LampTest(bool b);
338:
339:     BCD Select();           // Uses last value and state
340:
341:     BCD Select(
342:         enum AsmChannelWMSelect WMSelect,
343:         enum AsmChannelZonesSelect ZoneSelect,
344:         bool InvertSign,
345:         enum AsmChannelSignSelect SignSelect,
346:         enum AsmChannelNumericSelect NumSelect
347:     );
348:
349:     BCD Get();
350:
351:     BCD Set(BCD v) { value = v; valid = true; }
352:
353:     BCD Reset() {
354:         value = BCD::BCDConvert(BITC);
355:         valid = false;
356:         AsmChannelWMSelect = AsmChannelWMNone;
357:         AsmChannelZonesSelect = AsmChannelZonesNone;
358:         AsmChannelInvertSign = false;
359:         AsmChannelSignSelect = AsmChannelSignNone;
360:         AsmChannelNumericSelect = AsmChannelNumNone;
361:     }
362: };
363:
364: □
365:
366: // This class defines what is in an I/O Channel
367:
368: #define IOCHNOTREADY    1
369: #define IOCHBUSY        2
370: #define IOCHDATACHECK   4
371: #define IOCHCONDITION   8
372: #define IOCHNOTTRANSFER 16
373: #define IOCHWLRECORD    32
374:
375: #define IOLAMPNOTREADY  0
376: #define IOLAMPBUSY      1
377: #define IOLAMPDATACHECK 2
378: #define IOLAMPCONDITION 3
379: #define IOLAMPNOTTRANSFER 4
380: #define IOLAMPWLRECORD  5
381:
382: class T1410Channel : public TDisplayObject {
383:
384: public:
385:
386:     // Functions inherited from abstract base class classes now need definition
387:
388:     void OnComputerReset();
389:     void OnProgramReset();
390:     void Display();
391:     void LampTest(bool b);
392:
393: private:

```

*Gate A Channel To Assembly (b60)*  
*Gate B Reg To Assembly (b61)*  
*Moved to .app*

```
394:
395:     // Channel information
396:
397:     int ChStatus;                                // Channel status (see defines)
398:
399:     TLabel *ChStatusDisplay[6];                 // Channel status lights
400:
401: public:
402:
403:     TRegister *ChOp;
404:     TRegister *ChUnitType;
405:     TRegister *ChUnitNumber;
406:     TRegister *ChR1, *ChR2;
407:
408:     TDisplayLatch *ChInterlock;
409:     TDisplayLatch *ChRBCInterlock;
410:     TDisplayLatch *ChRead;
411:     TDisplayLatch *ChWrite;
412:     TDisplayLatch *ChOverlap;
413:     TDisplayLatch *ChNotOverlap;
414:
415:     enum TapeDensity {
416:         DENSITY_200_556 = 0, DENSITY_200_800 = 1, DENSITY_556_800 = 2
417:     } TapeDensity;
418:
419:
420:     // Methods
421:
422:     T1410Channel(                                     // Constructor
423:         TLabel *LampInterlock,
424:         TLabel *LampRBCInterlock,
425:         TLabel *LampRead,
426:         TLabel *LampWRIte,
427:         TLabel *LampOverlap,
428:         TLabel *LampNotOverlap,
429:         TLabel *LampNotRead,
430:         TLabel *LampBusy,
431:         TLabel *LampDataCheck,
432:         TLabel *LampCondition,
433:         TLabel *LampWLRecord,
434:         TLabel *LampNoTransfer
435:     );
436:
437:     inline int SetStatus(int i) { return ChStatus = i; }
438:     inline int GetStatus() { return ChStatus; }
439: };
440:
441:
442: □
443: // This class defines what is actually inside the CPU.
444:
445: #define MAXCHANNEL 2
446: #define CHANNEL1 0
447: #define CHANNEL2 1
448:
449: #define STORAGE 80000
450:
451: #define I_RING_OP 0
452: #define I_RING_1 1
453: #define I_RING_2 2
454: #define I_RING_3 3
455: #define I_RING_4 4
456: #define I_RING_5 5
457: #define I_RING_6 6
458: #define I_RING_7 7
459: #define I_RING_8 8
```

```
460: #define I_RING_9 9
461: #define I_RING_10 10
462: #define I_RING_11 11
463: #define I_RING_12 12
464:
465: #define A_RING_1 0
466: #define A_RING_2 1
467: #define A_RING_3 2
468: #define A_RING_4 3
469: #define A_RING_5 4
470: #define A_RING_6 5
471:
472: #define CLOCK_A 0
473: #define CLOCK_B 1
474: #define CLOCK_C 2
475: #define CLOCK_D 3
476: #define CLOCK_E 4
477: #define CLOCK_F 5
478: #define CLOCK_G 6
479: #define CLOCK_H 7
480: #define CLOCK_J 8
481: #define CLOCK_K 9
482:
483: #define SCAN_N 0
484: #define SCAN_1 1
485: #define SCAN_2 2
486: #define SCAN_3 3
487:
488: #define SUB_SCAN_NONE 0
489: #define SUB_SCAN_U 1
490: #define SUB_SCAN_B 2
491: #define SUB_SCAN_E 3
492: #define SUB_SCAN_MQ 4
493:
494: #define CYCLE_A 0
495: #define CYCLE_B 1
496: #define CYCLE_C 2
497: #define CYCLE_D 3
498: #define CYCLE_E 4
499: #define CYCLE_F 5
500: #define CYCLE_I 6
501: #define CYCLE_X 7
502:
503: class T1410CPU {
504:
505: private:
506:
507:     BCD core[STORAGE];
508:
509: public:
510:
511:     // Wiring list
512:
513:     TCpuObject *ResetList;           // List of latches.
514:     TDisplayObject *DisplayList;    // List of displayable things
515:
516:     // Data Registers
517:
518:     TRegister *A_Reg, *B_Reg, *Op_Reg, *Op_Mod_Reg;
519:
520:     // Address Registers
521:
522:     TAddressRegister *STAR;        // Storage Address Register
523:                                         // AKA MAR (Memory Address Register)
524:
525:     TAddressRegister *A_AR, *B_AR, *C_AR, *D_AR, *E_AR, *F_AR;
```

```

526:     TAddressRegister *I_AR;
527:
528:
529: // Channels
530:
531: T1410Channel *Channel[MAXCHANNEL];           // 2 I/O Channels.
532: TAChannel *AChannel;                         // A Channel
533:
534: // Indicators
535:
536: TDisplayIndicator *OffNormal;                // OFF NORMAL Indicator
537:
538: // Ring Counters
539:
540: TRingCounter *IRing;                         // Instruction decode ring
541: TRingCounter *ARing;                         // Address decode ring
542: TRingCounter *ClockRing;                     // Cycle Clock
543: TRingCounter *ScanRing;                      // Address Modification Mode
544: TRingCounter *SubScanRing;                   // Arithmetic Scan type
545: TRingCounter *CycleRing;                     // CPU Cycle type
546:
547: // Latches
548:
549:
550: TDisplayLatch *CarryIn;                      // Carry latch
551: TDisplayLatch *CarryOut;                     // Adder has generated carry
552: TDisplayLatch *AComplement;                  // A channel complement
553: TDisplayLatch *BComplement;                 // B channel complement
554: TDisplayLatch *CompareBGTA;                // B > A
555: TDisplayLatch *CompareBEQA;                // B = A
556: TDisplayLatch *CompareBLTA;                // B < A NOTE: On after C. Reset.
557: TDisplayLatch *Overflow;                    // Arithmetic Overflow
558: TDisplayLatch *DivideOverflow;              // Divide Overflow
559: TDisplayLatch *ZeroBalance;                 // Zero arithmetic result
560:
561: // Check Latches
562:
563: TDisplayLatch *AChannelCheck;                // A Channel parity error
564: TDisplayLatch *BChannelCheck;                // B Channel parity error
565: TDisplayLatch *AssemblyChannelCheck;         // Assembly Channel parity error
566: TDisplayLatch *AddressChannelCheck;          // Address Channel parity error
567: TDisplayLatch *AddressExitCheck;             // Validity error at address reg.
568: TDisplayLatch *ARegisterSetCheck;            // A register failed to reset
569: TDisplayLatch *BRegisterSetCheck;            // B register failed to reset
570: TDisplayLatch *OpRegisterSetCheck;           // Op register failed to set
571: TDisplayLatch *OpModifierSetCheck;           // Op modifier failed to set
572: TDisplayLatch *ACharacterSelectCheck;        // Incorrect A channel gating
573: TDisplayLatch *BCharacterSelectCheck;        // Incorrect B channel getting
574:
575: TDisplayLatch *IOInterlockCheck;              // Program did not check I/O
576: TDisplayLatch *AddressCheck;                 // Program gave bad address
577: TDisplayLatch *RBCInterlockCheck;             // Program did not check RBC
578: TDisplayLatch *InstructionCheck;              // Program issued invalide op
579:
580: // Switches
581:
582: enum Mode {                                     // Mode switch, values must match
583:     MODE_RUN = 0, MODE_DISPLAY = 1, MODE_ALTER = 2,
584:     MODE_CE = 3, MODE_TE = 4, MODE_ADDR = 5
585: } Mode;
586:
587: enum AddressEntry {
588:     ADDR_ENTRY_I = 0, ADDR_ENTRY_A = 1, ADDR_ENTRY_B = 2,
589:     ADDR_ENTRY_C = 3, ADDR_ENTRY_D = 4, ADDR_ENTRY_E = 5,
590:     ADDR_ENTRY_F = 6
591: } AddressEntry;

```

```
592:
593:     // The entry below is for the state of the 1415 Storage Scan SWITCH
594:     // (The storage scan modification mode is in the Ring ScanRing)
595:
596:     enum StorageScan {
597:         SSCAN_OFF = 0, SSCAN_LOAD_1 = 1, SSCAN_LOAD_0 = 2,
598:         SSCAN_REGEN_0 = 3, SSCAN_REGEN_1 = 4
599:     } StorageScan;
600:
601:     enum CycleControl {
602:         CYCLE_OFF = 0, CYCLE_LOGIC = 1, CYCLE_STORAGE = 2
603:     } CycleControl;
604:
605:     enum CheckControl {
606:         CHECK_STOP = 0, CHECK_RESTART = 1, CHECK_RESET = 2
607:     } CheckControl;
608:
609:     bool DiskWrInhibit;
610:     bool AsteriskInsert;
611:     bool InhibitPrintOut;
612:
613:     bool StopLatch;
614:     bool StopKeyLatch;
615:     bool DisplayModeLatch;
616:
617:     bool StorageWrapLatch;
618:
619:     BCD BitSwitches;
620:
621:     // Methods
622:
623:     T1410CPU();                                // Constructor
624:     void Display();                            // Run thru the display list
625:
626: private:
627:
628:     // Indicator Routines
629:     // Normally, these will be accessed via a bool (_closure *func)()
630:
631:     bool IndicatorOffNormal();
632:
633: public:
634:
635:     // Dynamic Channels
636:
637:     static BCD AssemblyChannel();
638:
639: public:
640:
641:     // Core operations (using STAR/MAR and B Data Register)
642:
643:     void Readout();                           // Reads out one storage character
644:     void Store();                            // Stores character in B Data Register
645:     void Store(BCD bcd);                     // Sets B Data Register, then stores
646:     void SetScan(char s);                   // Sets Scan Modification value
647:
648:     long STARScan();                        // Applies Scan modification to STAR,
649:                                         // and returns the results - suitable to assign
650:
651: };
652:
653: extern T1410CPU *CPU;
654:
655: //-----
656: #endif
657:
```

```
1: // This Unit provides the functionality behind some of my private
2: // types for the 1410 emulator
3:
4: -----
5: #include <vcl\vcl.h>
6: #include <assert.h>
7: #pragma hdrstop
8:
9: #include "UI1410CPU.h"
10: #include "UI1415L.h"
11: #include "UI1415CE.h"
12: #include "UI1410PWR.h"
13: #include "UI1410DEBUG.h"
14: #include "UI1410CPU.h"
15:
16: -----
17:
18: // Declarations for Borland VCL controls
19:
20: #include <vcl\Classes.hpp>
21: #include <vcl\Controls.hpp>
22: #include <vcl\StdCtrls.hpp>
23:
24: // Core pre-load (gives this virtual 1410 a 7010-like load capability
25: // Just hit Computer Reset then Start to boot from tape
26:
27: char *core_load_chars = "AL%BO00012$N..";
28: char core_load_wm[] = { 0,1,0,0,0,0,0,0,0,0,0,1,1,1 };
29:
30: // Tables which pre-multiply integers by decimal numbers, for efficiency.
31:
32: long ten_thousands[] = {
33:     0,10000,20000,30000,40000,50000,60000,70000,80000,90000
34: };
35:
36: long thousands[] = {
37:     0,1000,2000,3000,4000,5000,6000,7000,8000,9000
38: };
39:
40: long hundreds[] = {
41:     0,100,200,300,400,500,600,700,800,900
42: };
43:
44: long tens[] = { 0,10,20,30,40,50,60,70,80,90 };
45:
46: // Storage Scan modification values
47: // Correspond to the values in the enum StorageScan
48:
49: long scan_mod[] = { 0, -1, +1, -1 };
50:
51: // Implementation of TCpuObject (Abstract Base Class)
52:
53: // Everything in the CPU is on the reset list.
54: // Everything is reset by Computer Reset
55: // NOT Everything is reset by Program Reset!
56:
57: TCpuObject::TCpuObject()
58: {
59:     NextReset = CPU -> ResetList;
60:     CPU -> ResetList = this;
61: }
62:
63: // Implementation of TDisplayObject (Abstract Base Class)
64:
65: // What is special about these objects is that they are on the display list.
66:
```

```
67: TDisplayObject::TDisplayObject()
68: {
69:     NextDisplay = CPU -> DisplayList;
70:     CPU -> DisplayList = this;
71: }
72:
73: □
74:
75: // Implementation of TDisplayLatch (Display Latch Base Class)
76:
77: // The constructor initializes the latch and sets the pointer to a lamp.
78: // A pointer to a lamp is required.
79:
80: // Default is to be reset by a Program Reset, but this can be overridden
81: // when the constructor is called, if necessary.
82:
83: TDisplayLatch::TDisplayLatch(TLabel *l)
84: {
85:     state = false;
86:     doprogramreset = true;
87:     lamp = l;
88: }
89:
90: // The second constructor does the same thing, but is passed a variable
91: // which indicates whether or not the latch should be reset by Program
92: // Reset.
93:
94: TDisplayLatch::TDisplayLatch(TLabel *l, bool progreset)
95: {
96:     state = false;
97:     doprogramreset = progreset;
98:     lamp = l;
99: }
100:
101: // All Displayable Latches are reset on a COMPUTER RESET
102:
103: void TDisplayLatch::OnComputerReset()
104: {
105:     Reset();
106: }
107:
108: // Whether or not a displayable latch is reset by program reset depends
109: // on the latch.
110:
111: void TDisplayLatch::OnProgramReset()
112: {
113:     if(doprogramreset) {
114:         Reset();
115:     }
116: }
117:
118: // Routine to set a latch and set the CPU Stop Latch at the same time
119: // Typically used for error latches.
120:
121: void TDisplayLatch::SetStop() {
122:     state = true;
123:     CPU -> StopLatch = true;
124: }
125:
126: // When the Display routine is called, it sets or resets the lamp,
127: // depending on the current state.
128:
129: void TDisplayLatch::Display()
130: {
131:     lamp -> Enabled = state;
132:     lamp -> Repaint();
```

```
133: }
134:
135: // On a lamp test, light all the lamp.
136: // On reset of a lamp test, display the current state.
137:
138: void TDisplayLatch::LampTest(bool b)
139: {
140:     lamp -> Enabled = (b ? true : state);
141:     lamp -> Repaint();
142: }
143:
144: □
145:
146: // Implementation of TRingCounter (Ring Counter Class)
147:
148: // The constructor sets the max state of the ring, and resets the ring
149: // It also allocates an array of pointers to the lamps. The creator must
150: // fill in that array, however. Initially, the lamp pointers are empty,
151: // (which means no lamp is attached to that state).
152:
153: TRingCounter::TRingCounter(char n)
154: {
155:     int i;
156:
157:     state = 0;
158:     max = n;
159:     lastlamp = 0;
160:
161:     lamps = new TLabel*[n];
162:     for(i=0; i < n; ++i) {
163:         lamps[i] = 0;
164:     }
165: }
166:
167: // The destructor frees up the array of lamps. Probably will never use.
168:
169: __fastcall TRingCounter::~TRingCounter()
170: {
171:     delete[] lamps;
172: }
173:
174: // All Ring counters are reset on PROGRAM and COMPUTER RESET
175:
176: void TRingCounter::OnComputerReset()
177: {
178:     state = 0;
179: }
180:
181: void TRingCounter::OnProgramReset()
182: {
183:     state = 0;
184: }
185:
186: // When the Display routine is called, it resets the lamp corresponding
187: // to the state when it last displayed, and then displays the current state
188: // If there is no lamp associated with a state (lamp pointer is null),
189: // then don't display any lamp.
190:
191: void TRingCounter::Display()
192: {
193:     if(lastlamp) {
194:         lastlamp -> Enabled = false;
195:         lastlamp -> Repaint();
196:     }
197:     if(lamps[state] == 0) {
198:         lastlamp = 0;
```

```
199:         return;
200:     }
201:     lamps[state] -> Enabled = true;
202:     lamps[state] -> Repaint();
203:     lastlamp = lamps[state];
204: }
205:
206: // On a lamp test, light all the associated lamps.
207: // On reset of a lamp test, clear them all, then display the current state.
208:
209: void TRingCounter::LampTest(bool b)
210: {
211:     int i;
212:
213:     for(i = 0; i < max; ++i) {
214:         if(lamps[i] != 0) {
215:             lamps[i] -> Enabled = b;
216:             lamps[i] -> Repaint();
217:         }
218:     }
219:     if(!b) {
220:         lastlamp = 0;
221:         this -> Display();
222:     }
223: }
224:
225: // Next advances to the next state (or back to the start if appropriate)
226: // Returns current state. This is only useful for true Ring counters.
227:
228: char TRingCounter::Next()
229: {
230:     if(++state >= max) {
231:         state = 0;
232:     }
233:     return(state);
234: }
235:
236: □
237:
238: // Implementation of Simple Registers
239:
240: // Assignment: Just assign the value, NOT the things from TCpuObject!!
241: // (Those need to stay unchanged as they should be invariant once the
242: // register is created: it's position on the reset list and whether or
243: // not it is affected by Program Reset, for example.
244:
245: void TRegister::operator=(TRegister &source)
246: {
247:     value = source.value;
248: }
249:
250: □
251:
252: // Implementation of Address Registers
253:
254: // The constructor just initializes things so that we know that the
255: // address register contains an invalid value.
256:
257: TAddressRegister::TAddressRegister()
258: {
259:     i_valid = false;
260:     i_value = 0;
261:     d_valid = false;
262:     set[0] = set[1] = set[2] = set[3] = set[4] = false;
263:     DoesProgramReset = false;
264: }
```

```
265:
266: // A function to determine whether or not the address register contains
267: // a valid value.
268:
269: bool TAddressRegister::IsValid()
270: {
271:     if(i_valid || d_valid ||
272:         (set[0] && set[1] && set[2] && set[3] && set[4]) ) {
273:         return(true);
274:     }
275:     return(false);
276: }
277:
278: // A routine to reset an address register: to binary 0. Note that this
279: // means it is invalid (i.e. will cause an Address Exit Check if you try
280: // and actually use the value except to print it out on the console.
281:
282: void TAddressRegister::Reset()
283: {
284:     TWOOF5 zero;
285:
286:     i_valid = false;
287:     d_valid = false;
288:     set[0] = set[1] = set[2] = set[3] = set[4] = false;
289:     digits[0] = digits[1] = digits[2] = digits[3] = digits[4] = zero;
290:     i_value = 0;
291: }
292:
293: // A routine to get the value of an address register. Note that if
294: // the value is invalid, the result is an Address Exit Check.
295:
296: long TAddressRegister::Gate()
297: {
298:     if(i_valid) {
299:         return(i_value);
300:     }
301:     else if(IsValid()) {
302:         i_value = ten_thousands[digits[0].ToInt()] +
303:             thousands[digits[1].ToInt()] + hundreds[digits[2].ToInt()] +
304:             tens[digits[3].ToInt()] + digits[4].ToInt();
305:         i_valid = true;
306:         d_valid = true;
307:         return(i_value);
308:     }
309:     else {
310:         CPU -> AddressExitCheck -> SetStop();           // Address exit error!
311:         return(-1);
312:     }
313: }
314:
315: // Get a single character from an address register. This is valid even
316: // if the value in the register is invalid.
317:
318: BCD TAddressRegister::GateBCD(int i)
319: {
320:     if(d_valid) {
321:         if(digits[i-1].ToInt() < 0) {
322:             CPU -> AddressExitCheck -> SetStop();
323:             return(0);
324:         }
325:         return(digits[i-1].ToBCD());
326:     }
327:     else if(IsValid()) {
328:         digits[4] = i_value % 10;
329:         digits[3] = i_value % 100 - digits[4].ToInt();
330:         digits[2] = i_value % 1000 - (10*digits[3].ToInt()) - digits[4].ToInt();
```

```
331:         digits[1] = i_value % 10000 - (100*digits[2].ToInt()) -
332:             (10*digits[3].ToInt()) - digits[4].ToInt();
333:         digits[0] = i_value - (1000*digits[1].ToInt()) -
334:             (100*digits[2].ToInt()) - (10*digits[3].ToInt()) - digits[4].ToInt();
335:         d_valid = true;
336:         set[0] = set[1] = set[2] = set[3] = set[4] = true;
337:         if(digits[i-1].ToInt() < 0) {
338:             CPU -> AddressExitCheck -> SetStop();
339:             return(0);
340:         }
341:         return(digits[i-1].ToBCD());
342:     }
343:     else {
344:         CPU -> AddressExitCheck -> SetStop();
345:         return(BCD(0));
346:     }
347: }
348:
349: // A routine to set a single digit of an address register.
350: // This effectively implements the address channel.
351:
352: void TAddressRegister::Set(TWOOF5 digit,int i)
353: {
354:     if(digit.ToInt() == -1) {
355:         CPU -> AddressChannelCheck -> SetStop();
356:     }
357:
358:     digits[i-1] = digit;
359:     set[i-1] = true;
360:     if(i == 5 && set[0] && set[1] && set[2] && set[3]) {
361:         d_valid = true;
362:     }
363: }
364:
365: // A routine to set an address register from a binary value. Should
366: // really only be used to reset IAR to 1 during a reset operation.
367:
368: void TAddressRegister::Set(long i)
369: {
370:     d_valid = false;
371:     set[0] = set[1] = set[2] = set[3] = set[4] = false;
372:     i_valid = true;
373:     i_value = i;
374: }
375:
376: // Assignment: Just assign the value, NOT the things from TCpuObject!!
377: // (Those need to stay unchanged as they should be invariant once the
378: // register is created: it's position on the reset list and whether or
379: // not it is affected by Program Reset, for example.
380:
381: // Note that an attempt to assign from an invalid register does do the
382: // set, but also sets the Address Exit Check error in the CPU
383:
384: void TAddressRegister::operator=(TAddressRegister &source)
385: {
386:     int i;
387:
388:     i_value = source.i_value;
389:     i_valid = source.i_valid;
390:     d_valid = source.d_valid;
391:     if(!source.IsValid()) {
392:         CPU -> AddressExitCheck -> SetStop();
393:     }
394:     for(i=0; i++ < 5) { i < 5; ++i ! !
395:         digits[i] = source.digits[i];
396:         set[i] = source.set[i];
```

```
397:     }
398: }
399:
400:
401: □
402:
403: // Implementation of I/O Channel Class
404:
405: // Constructor.  Initializes state
406:
407: T1410Channel::T1410Channel(
408:     TLabel *LampInterlock,
409:     TLabel *LampRBCInterlock,
410:     TLabel *LampRead,
411:     TLabel *LampWrite,
412:     TLabel *LampOverlap,
413:     TLabel *LampNotOverlap,
414:     TLabel *LampNotReady,
415:     TLabel *LampBusy,
416:     TLabel *LampDataCheck,
417:     TLabel *LampCondition,
418:     TLabel *LampWLRecord,
419:     TLabel *LampNoTransfer ) {
420:
421:     ChStatus = 0;
422:     TapeDensity = DENSITY_200_556;
423:
424:     ChStatusDisplay[IOLAMPNOTREADY] = LampNotReady;
425:     ChStatusDisplay[IOLAMPBUSY] = LampBusy;
426:     ChStatusDisplay[IOLAMPDATACHECK] = LampDataCheck;
427:     ChStatusDisplay[IOLAMPCONDITION] = LampCondition;
428:     ChStatusDisplay[IOLAMPNOTTRANSFER] = LampNoTransfer;
429:     ChStatusDisplay[IOLAMPWLRECORD] = LampWLRecord;
430:
431:     // Generally, the channel latches are *not* reset by Program Reset
432:
433:     ChInterlock = new TDisplayLatch(LampInterlock, false);
434:     ChRBCInterlock = new TDisplayLatch(LampRBCInterlock, false);
435:     ChRead = new TDisplayLatch(LampRead, false);
436:     ChWrite = new TDisplayLatch(LampWrite, false);
437:     ChOverlap = new TDisplayLatch(LampOverlap, false);
438:     ChNotOverlap = new TDisplayLatch(LampNotOverlap, false);
439:
440:     // Generally, the channel registers are *not* reset by Program Reset
441:
442:     ChOp = new TRegister(false);
443:     ChUnitType = new TRegister(false);
444:     ChUnitNumber = new TRegister(false);
445:     ChR1 = new TRegister(false);
446:     ChR2 = new TRegister(false);
447: }
448:
449: // Channel is reset during ComputerReset
450:
451: void T1410Channel::OnComputerReset()
452: {
453:     ChStatus = 0;
454:
455:     // Note: The objects which are TDisplayLatch objects will reset themselves
456: }
457:
458: // Channel is not reset during Program Reset
459:
460: void T1410Channel::OnProgramReset()
461: {
462:     // Channel not affected by Program Reset
```

```
463: }
464:
465: //  Display Routine.
466:
467: void T1410Channel::Display() {
468:
469:     int i;
470:
471:     ChStatusDisplay[IOLAMPNOTREADY] -> Enabled =
472:         ((ChStatus & IOCHNOTREADY) != 0);
473:     ChStatusDisplay[IOLAMPBUSY] -> Enabled =
474:         ((ChStatus & IOCHBUSY) != 0);
475:     ChStatusDisplay[IOLAMPDATACHECK] -> Enabled =
476:         ((ChStatus & IOCHDATACHECK) != 0);
477:     ChStatusDisplay[IOLAMPCONDITION] -> Enabled =
478:         ((ChStatus & IOCHCONDITION) != 0);
479:     ChStatusDisplay[IOLAMPWLRECORD] -> Enabled =
480:         ((ChStatus & IOCHWLRECORD) != 0);
481:     ChStatusDisplay[IOLAMPNOTTRANSFER] -> Enabled =
482:         ((ChStatus & IOCHNOTTRANSFER) != 0);
483:
484:     for(i=0; i <= 5; ++i) {
485:         ChStatusDisplay[i] -> Repaint();
486:     }
487:
488: //  Although in most instances the following would be redundant,
489: //  because these objects are also on the CPU display list, we include
490: //  them here in case we want to display a channel separately.
491:
492:     ChInterlock -> Display();
493:     ChRBCTInterlock -> Display();
494:     ChRead -> Display();
495:     ChWrite -> Display();
496:     ChOverlap -> Display();
497:     ChNotOverlap -> Display();
498: }
499:
500: //  Channel Lamp Test
501:
502: void T1410Channel::LampTest(bool b)
503: {
504:     int i;
505:
506:     // Note, we don't have to do anything to the TDisplayLatch objects in
507:     // the channel for lamp test.  They will take care of themselves on a
508:     // lamp test.
509:
510:     if(!b) {
511:         for(i=0; i <= 5; ++i) {
512:             ChStatusDisplay[i] -> Enabled = true;
513:             ChStatusDisplay[i] -> Repaint();
514:         }
515:     }
516:     else {
517:         Display();
518:     }
519: }
520:
521: □
522:
523: //  CPU object constructor.  Essentially this method "wires" the 1410.
524:
525: T1410CPU::T1410CPU()
526: {
527:     long i;
528:
```

```
529:     CPU = this;
530:
531: // Clear out the lists
532:
533: DisplayList = 0;
534: ResetList = 0;
535:
536: // Set switches to initial states
537:
538: Mode = MODE_RUN;
539: AddressEntry = ADDR_ENTRY_I;
540: StorageScan = SSCAN_OFF;
541: CycleControl = CYCLE_OFF;
542: CheckControl = CHECK_STOP;
543: DiskWrInhibit = false;
544: AsteriskInsert = true;
545: InhibitPrintOut = false;
546: BitSwitches = BCD(0);
547:
548: // Build the various displayable components of the CPU
549:
550: IRing = new TRingCounter(13);
551: assert(F1415L -> Light_I_OP != 0);
552: IRing -> lamps[0] = F1415L -> Light_I_OP;
553: IRing -> lamps[1] = F1415L -> Light_I_1;
554: IRing -> lamps[2] = F1415L -> Light_I_2;
555: IRing -> lamps[3] = F1415L -> Light_I_3;
556: IRing -> lamps[4] = F1415L -> Light_I_4;
557: IRing -> lamps[5] = F1415L -> Light_I_5;
558: IRing -> lamps[6] = F1415L -> Light_I_6;
559: IRing -> lamps[7] = F1415L -> Light_I_7;
560: IRing -> lamps[8] = F1415L -> Light_I_8;
561: IRing -> lamps[9] = F1415L -> Light_I_9;
562: IRing -> lamps[10] = F1415L -> Light_I_10;
563: IRing -> lamps[11] = F1415L -> Light_I_11;
564: IRing -> lamps[12] = F1415L -> Light_I_12;
565:
566: ARing = new TRingCounter(6);
567: ARing -> lamps[0] = F1415L -> Light_A_1;
568: ARing -> lamps[1] = F1415L -> Light_A_2;
569: ARing -> lamps[2] = F1415L -> Light_A_3;
570: ARing -> lamps[3] = F1415L -> Light_A_4;
571: ARing -> lamps[4] = F1415L -> Light_A_5;
572: ARing -> lamps[5] = F1415L -> Light_A_6;
573:
574: ClockRing = new TRingCounter(10);
575: ClockRing -> lamps[0] = F1415L -> Light_Clk_A;
576: ClockRing -> lamps[1] = F1415L -> Light_Clk_B;
577: ClockRing -> lamps[2] = F1415L -> Light_Clk_C;
578: ClockRing -> lamps[3] = F1415L -> Light_Clk_D;
579: ClockRing -> lamps[4] = F1415L -> Light_Clk_E;
580: ClockRing -> lamps[5] = F1415L -> Light_Clk_F;
581: ClockRing -> lamps[6] = F1415L -> Light_Clk_G;
582: ClockRing -> lamps[7] = F1415L -> Light_Clk_H;
583: ClockRing -> lamps[8] = F1415L -> Light_Clk_J;
584: ClockRing -> lamps[9] = F1415L -> Light_Clk_K;
585:
586: ScanRing = new TRingCounter(4);
587: ScanRing -> lamps[0] = F1415L -> Light_Scan_N;
588: ScanRing -> lamps[1] = F1415L -> Light_Scan_1;
589: ScanRing -> lamps[2] = F1415L -> Light_Scan_2;
590: ScanRing -> lamps[3] = F1415L -> Light_Scan_3;
591:
592: SubScanRing = new TRingCounter(5);
593: // NOTE: State 0 is "OFF" - no flip flops set
594: SubScanRing -> lamps[1] = F1415L -> Light_Sub_Scan_U;
```

```
595:     SubScanRing -> lamps[2] = F1415L -> Light_Sub_Scan_B;
596:     SubScanRing -> lamps[3] = F1415L -> Light_Sub_Scan_E;
597:     SubScanRing -> lamps[4] = F1415L -> Light_Sub_Scan_MQ;
598:
599:     CycleRing = new TRingCounter(8);
600:     CycleRing -> lamps[0] = F1415L -> Light_Cycle_A;
601:     CycleRing -> lamps[1] = F1415L -> Light_Cycle_B;
602:     CycleRing -> lamps[2] = F1415L -> Light_Cycle_C;
603:     CycleRing -> lamps[3] = F1415L -> Light_Cycle_D;
604:     CycleRing -> lamps[4] = F1415L -> Light_Cycle_E;
605:     CycleRing -> lamps[5] = F1415L -> Light_Cycle_F;
606:     CycleRing -> lamps[6] = F1415L -> Light_Cycle_I;
607:     CycleRing -> lamps[7] = F1415L -> Light_Cycle_X;
608:
609: // Build the various latches. Most of these are not
610: // reset during Program Reset
611:
612: StopLatch = true;
613: StopKeyLatch = false;
614: DisplayModeLatch = false;
615:
616: CarryIn = new TDisplayLatch(F1415L -> Light_Carry_In, false);
617: CarryOut = new TDisplayLatch(F1415L -> Light_Carry_Out, false);
618: AComplement = new TDisplayLatch(F1415L -> Light_A_Complement, false);
619: BComplement = new TDisplayLatch(F1415L -> Light_B_Complement, false);
620:
621: CompareBGTA = new TDisplayLatch(F1415L -> Light_B_GT_A, false);
622: CompareBEQA = new TDisplayLatch(F1415L -> Light_B_EQ_A, false);
623: CompareBLTA = new TDisplayLatch(F1415L -> Light_B_LT_A, false);
624: Overflow = new TDisplayLatch(F1415L -> Light_Overflow, false);
625: DivideOverflow = new TDisplayLatch(F1415L -> Light_Divide_Overflow, false);
626: ZeroBalance = new TDisplayLatch(F1415L -> Light_Zero_Balance, false);
627:
628: // Build the various check latches. Program Reset does reset these
629:
630: AChannelCheck = new TDisplayLatch(F1415L -> Light_Check_AChannel);
631: BChannelCheck = new TDisplayLatch(F1415L -> Light_Check_BChannel);
632: AssemblyChannelCheck = new TDisplayLatch(F1415L ->
Light_Check_AssemblyChannel);
633: AddressChannelCheck = new TDisplayLatch(F1415L ->
Light_Check_AddressChannel);
634: AddressExitCheck = new TDisplayLatch(F1415L -> Light_Check_AddressExit);
635: ARegisterSetCheck = new TDisplayLatch(F1415L -> Light_Check_ARegisterSet);
636: BRegisterSetCheck = new TDisplayLatch(F1415L -> Light_Check_BRegisterSet);
637: OpRegisterSetCheck = new TDisplayLatch(F1415L -> Light_Check_OpRegisterSet);
638: OpModifierSetCheck = new TDisplayLatch(F1415L -> Light_Check_OpModifierSet);
639: ACharacterSelectCheck = new TDisplayLatch(F1415L ->
Light_Check_ACharacterSelect);
640: BCharacterSelectCheck = new TDisplayLatch(F1415L ->
Light_Check_BCharacterSelect);
641:
642: IOInterlockCheck = new TDisplayLatch(F1415L -> Light_Check_IOInterlock);
643: AddressCheck = new TDisplayLatch(F1415L -> Light_Check_AddressCheck);
644: RBCInterlockCheck = new TDisplayLatch(F1415L -> Light_Check_IOInterlock);
645: InstructionCheck = new TDisplayLatch(F1415L -> Light_Check_InstructionCheck);
646:
647: // Build the Data Registers
648:
649: A_Reg = new TRegister();
650: B_Reg = new TRegister();
651: Op_Reg = new TRegister(0);
652: Op_Mod_Reg = new TRegister(0);
653:
654: // Build the Address Registers
655:
656: STAR = new TAddressRegister();
```

```
657:     A_AR = new TAddressRegister();
658:     B_AR = new TAddressRegister();
659:     C_AR = new TAddressRegister();
660:     D_AR = new TAddressRegister();
661:     E_AR = new TAddressRegister();
662:     F_AR = new TAddressRegister();
663:     I_AR = new TAddressRegister();
664:
665: // Build the channels
666:
667: Channel[CHANNEL1] = new T1410Channel(
668:     F1415L -> Light_Ch1_Interlock,
669:     F1415L -> Light_Ch1_RBCInterlock,
670:     F1415L -> Light_Ch1_Read,
671:     F1415L -> Light_Ch1_Write,
672:     F1415L -> Light_Ch1_Overlap,
673:     F1415L -> Light_Ch1_NoOverlap,
674:     F1415L -> Light_Ch1_NotReady,
675:     F1415L -> Light_Ch1_Busy,
676:     F1415L -> Light_Ch1_DataCheck,
677:     F1415L -> Light_Ch1_Condition,
678:     F1415L -> Light_Ch1_WLRecord,
679:     F1415L -> Light_Ch1_NoTransfer
680: );
681:
682: Channel[CHANNEL2] = new T1410Channel(
683:     F1415L -> Light_Ch2_Interlock,
684:     F1415L -> Light_Ch2_RBCInterlock,
685:     F1415L -> Light_Ch2_Read,
686:     F1415L -> Light_Ch2_Write,
687:     F1415L -> Light_Ch2_Overlap,
688:     F1415L -> Light_Ch2_NoOverlap,
689:     F1415L -> Light_Ch2_NotReady,
690:     F1415L -> Light_Ch2_Busy,
691:     F1415L -> Light_Ch2_DataCheck,
692:     F1415L -> Light_Ch2_Condition,
693:     F1415L -> Light_Ch2_WLRecord,
694:     F1415L -> Light_Ch2_NoTransfer
695: );
696:
697: // Some latches are set after power on...
698:
699: CompareBLTA -> Set();
700: I_AR -> Set(1);
701:
702: // Initialize core-load
703:
704: for(i=0; i < strlen(core_load_chars); ++i) {
705:     core[i] = BCD::BCDConvert(core_load_chars[i]);
706:     if(core_load_wm[i]) {
707:         core[i].SetWM();
708:     }
709: }
710:
711: // Finally, set up the indicators
712:
713: OffNormal = new TDisplayIndicator(F1415L -> Light_Off_Normal,
714:                                     &(CPU -> IndicatorOffNormal));
715:
716: }
717:
718:
719: // CPU Object display - run thru the display list
720:
721: void T1410CPU::Display()
722: {
```

```
723:     TDisplayObject *l;
724:
725:     for(l = DisplayList; l != 0; l = l -> NextDisplay) {
726:         l -> Display();
727:     }
728: }
729:
730: // Off Normal Indicator routine
731:
732: bool T1410CPU::IndicatorOffNormal()
733: {
734:     return(CPU -> InhibitPrintOut ||
735:           !(CPU -> AsteriskInsert) ||
736:           CPU -> CycleControl != CPU -> CYCLE_OFF ||
737:           CPU -> CheckControl != CPU -> CHECK_STOP ||
738:           (CPU -> Mode == CPU -> MODE_CE &&
739:            CPU -> StorageScan != CPU -> SSCAN_OFF) ||
740:           CPU -> AddressEntry != CPU -> ADDR_ENTRY_I );
741: }
742:
743: // Assembly Channel (Dummy, for now)
744:
745: BCD T1410CPU::AssemblyChannel()
746: {
747:     return(BITC);
748: }
749:
750: // Storage routine to read out (access) core storage.
751: // (A real core storage unit would also have to store it back)
752: // Address is in the STAR (aka MAR)
753:
754: void T1410CPU::Readout()
755: {
756:     long i;
757:
758:     i = STAR -> Gate();           // Hee hee ;-)
759:     A Funny
760:
761:     if(i < 0) {
762:         DEBUG("Invalid address in STAR: Address Exit Check",0)
763:         AddressExitCheck -> SetStop();
764:         B_Reg -> Set(0);          // Force a B Channel Check
765:         return;
766:     }
767:
768:     if(i >= STORAGE) {
769:         i = 0;
770:         if(!DisplayModeLatch) {
771:             AddressCheck -> SetStop();
772:             B_Reg -> Set(0);
773:             return;
774:         }
775:         CPU -> StopKeyLatch = true;
776:     }
777:
778:     B_Reg -> Set(core[i]);
779:
780: // Storage routine to store what is in the B Data Register
781:
782: void T1410CPU::Store()
783: {
784:     long i;
785:
786:     i = STAR -> Gate();
787:
788:     if(i < 0) {
```

```
789:         DEBUG("Invalid address in STAR: Address Exit Check",0)
790:         AddressExitCheck -> SetStop();
791:         B_Reg -> Set(0);           // Force a B Channel Check
792:         return;
793:     }
794:
795:     if(i >= STORAGE) {
796:         i = 0;
797:         if(!DisplayModeLatch) {
798:             AddressCheck -> SetStop();
799:             B_Reg -> Set(0);
800:             return;
801:         }
802:         CPU -> StopKeyLatch = true;
803:     }
804:
805:     if(!B_Reg -> Get().CheckParity()) {
806:         BChannelCheck -> SetStop();
807:     }
808:     core[i] = B_Reg -> Get();
809: }
810:
811: // Storage routine to store data (Assembly Channel is implied here)
812:
813: void T1410CPU::Store(BCD bcd)
814: {
815:     B_Reg -> Set(bcd);
816:     if(!B_Reg -> Get().CheckParity()) {
817:         AssemblyChannelCheck -> SetStop();
818:     }
819:     Store();
820: }
821:
822: // Set Storage Scan Mode
823:
824: void T1410CPU::SetScan(char i)
825: {
826:     ScanRing -> Set(i);
827: }
828:
829: // Apply storage scan value to STAR and return result
830:
831: long T1410CPU::STARScan()
832: {
833:     return(STAR -> Gate() + scan_mod[ScanRing -> State()]);
834: }
```

```
1: #ifndef UBCDH
2: #define UBCDH
3:
4: //
5: //  Definition of character representation
6: //
7:
8: #define BITWM 0x80
9: #define BIT1 1
10: #define BIT2 2
11: #define BIT4 4
12: #define BIT8 8
13: #define BITA 0x10
14: #define BITB 0x20
15: #define BITC 0x40
16:
17: extern char bcd_ascii[];
18: extern int ascii_bcd[];
19: extern int bcd_to_two_of_five_table[];
20: extern int two_of_five_to_bin_table[];
21: extern int bin_to_two_of_five_table[];
22: extern int parity_table[];
23:
24: //
25: //  These are defined so that keyboard input may recognize them
26: //
27:
28: #define ASCII_RECORD_MARK    0174
29: #define ASCII_GROUP_MARK     0316
30: #define ASCII_SEGMENT_MARK   0327
31: #define ASCII_RADICAL        0373
32: #define ASCII_ALT_BLANK      'b'
33: #define ASCII_WORD_SEPARATOR '^'
34: #define ASCII_DELTA          0177
35:
36: class BCD {
37:
38: private:
39:     int c;                                // WM C B A 8 4 2 1
40:                                         // (Check bit is not used) X
41: public:
42:     BCD() { c = 0; }                      // Default constructor
43:
44:     BCD(int i) { c = i; }
45:
46:     static inline int BCDConvert(int ch) {
47:         if(ascii_bcd[ch] < 0) {
48:             return ascii_bcd[ASCII_ALT_BLANK];
49:         }
50:         else {
51:             return ascii_bcd[ch];
52:         }
53:     }
54:
55:     static inline int BCDCheck(int ch) {
56:         return(ascii_bcd[ch]);
57:     }
58:
59:     inline char ToAscii() {
60:         return bcd_ascii[c & 077];
61:     }
62:
63:     inline intToInt() {
64:         return c;
65:     }
66:
```

```

67:     inline bool TestWM() {
68:         return((c & BITWM) != 0);           Clear WM
69:     }
70:
71:     inline bool TestCheck() {
72:         return((c & BITC) != 0);           Complement Check
73:     }
74:
75:     inline void SetWM() {
76:         c |= BITWM;                      Clear Check
77:     }
78:
79:     inline void SetCheck() {
80:         c |= BITC;                      Check Parity
81:     }
82:
83: // Routine to test the parity of a BCD character
84: // Returns 0 for Even Parity, 1 for Odd Parity
85: // The test includes the WM bit, but does NOT itself
86: // include the check bit.
87:
88:     inline int GetParity() { return(parity_table[c]); } Set Odd Parity
89:
90: };
91:
92: #define TWOOF5_1      1
93: #define TWOOF5_2      2
94: #define TWOOF5_4      4
95: #define TWOOF5_8      8
96: #define TWOOF5_0     16
97:
98: class TWOOF5 {
99:
100: private:
101:     int b;
102:
103: public:
104:
105:     TWOOF5() { b = 0; }
106:     TWOOF5(BCD bcd) { b = bcd_to_two_of_five_table[bcd.ToInt() & 0x3f]; }
107:     TWOOF5(int i) { b = bin_to_two_of_five_table[i]; }
108:     inlineToInt() { return two_of_five_to_bin_table[b]; }
109:     inline ToBCD() {
110:         return(ascii_bcd['0' +ToInt()]);
111:     };
112: };
113:
114: #endif
115:

```

```
1: //  
2: // This Unit provides the implementation of bcd characters  
3: //  
4:  
5: #include "ubcd.h"  
6:  
7: /*  
8:     The following tables were copied from Joseph  
9:     Newcomer's 1401 emulation, in order to provide  
10:    consistent card, tape and print facilities.  
11:  
12:    Jay Jaeger, 12/96  
13: */  
14:  
15: /* The following table is given in the order of the 1401 BCD codes, and  
16:    contains the equivalent ASCII codes for printout.  
17: */  
18: char bcd_ascii[64] = {  
19:     ' ', /* 0           - space */  
20:     '1', /* 1           1   - 1 */  
21:     '2', /* 2           2   - 2 */  
22:     '3', /* 3           21  - 3 */  
23:     '4', /* 4           4   - 4 */  
24:     '5', /* 5           41  - 5 */  
25:     '6', /* 6           42  - 6 */  
26:     '7', /* 7           421 - 7 */  
27:     '8', /* 8           8   - 8 */  
28:     '9', /* 9           8 1 - 9 */  
29:     '0', /* 10          8 2 - 0 */  
30:     '=', /* 11          8 21 - number sign (#) or equal */  
31:     '\'', /* 12          84  - at sign @ or quote */  
32:     ':', /* 13          84 1 - colon */  
33:     '>', /* 14          842 - greater than */  
34:     'û', /* 15          8421 - radical */  
35:     'b', /* 16          A   - substitute blank */  
36:     '/', /* 17          A 1 - slash */  
37:     'S', /* 18          A 2 - S */  
38:     'T', /* 19          A 21 - T */  
39:     'U', /* 20          A 4 - U */  
40:     'V', /* 21          A 4 1 - V */  
41:     'W', /* 22          A 42 - W */  
42:     'X', /* 23          A 421 - X */  
43:     'Y', /* 24          A8  - Y */  
44:     'Z', /* 25          A8 1 - Z */  
45:     '\174', /* 26          A8 2 - record mark */  
46:     ',', /* 27          A8 21 - comma */  
47:     '(', /* 28          A84 - percent % or paren */  
48:     '^', /* 29          A84 1 - word separator */  
49:     '\\', /* 30          A842 - left oblique */  
50:     'x', /* 31          A8421 - segment mark */  
51:     '-', /* 32          B   - hyphen */  
52:     'J', /* 33          B 1 - J */  
53:     'K', /* 34          B 2 - K */  
54:     'L', /* 35          B 21 - L */  
55:     'M', /* 36          B 4 - M */  
56:     'N', /* 37          B 4 1 - N */  
57:     'O', /* 38          B 42 - O */  
58:     'P', /* 39          B 421 - P */  
59:     'Q', /* 40          B 8 - Q */  
60:     'R', /* 41          B 8 1 - R */  
61:     '!', /* 42          B 8 2 - exclamation (-0) */  
62:     '$', /* 43          B 8 21 - dollar sign */  
63:     '*', /* 44          B 84 - asterisk */  
64:     ']', /* 45          B 84 1 - right bracket */  
65:     ';', /* 46          B 842 - semicolon */  
66:     '\177', /* 47          B 8421 - delta */
```

```

67:     '+' ,    /* 48 BA      - ampersand or plus */
68:     'A' ,    /* 49 BA 1   - A */
69:     'B' ,    /* 50 BA 2   - B */
70:     'C' ,    /* 51 BA .21 - C */
71:     'D' ,    /* 52 BA 4   - D */
72:     'E' ,    /* 53 BA 4 1 - E */
73:     'F' ,    /* 54 BA 42  - F */
74:     'G' ,    /* 55 BA 421 - G */
75:     'H' ,    /* 56 BA8   - H */
76:     'I' ,    /* 57 BA8 1  - I */
77:     '?' ,    /* 58 BA8 2  - question mark */
78:     '.' ,    /* 59 BA8 21 - period */
79:     ')' ,    /* 60 BA84  - lozenge or paren */
80:     '[' ,    /* 61 BA84 1 - left bracket */
81:     '<' ,   /* 62 BA842 - less than */
82:     '†' ,    /* 63 BA8421 - group mark */

83: };
84:
85:
86: ****
87: The following table is used to convert ASCII characters to BCD.
88:
89: Note that it currently is not complete.
90:
91: The following substitutions or alternate mappings are made:
92:
93:     ASCII code   BCD      Notes
94:     -----   ---      -----
95:     "        N/A      illegal
96:     %        (        'H' character set representation
97:     &       +        'H' character set representation
98:     @        '        'H' character set representation
99:     #        =        'H' character set representation
100:    "        N/A      illegal
101:    ^        ^        substitute graphic for word-separator
102:    `        N/A      illegal
103:    a,c-z    A,C-Z    case folded
104:    b        b        substitute blank
105:    {        N/A      illegal
106:    }        N/A      illegal
107:    ~        N/A      illegal
108:    |        Ø        substitute for record mark
109:
110: ****
111:
112: int ascii_bcd[256] = {
113:     -1,-1,-1,-1,-1,-1,-1,-1,    /* 00 - 07 illegal */
114:     -1,-1,-1,-1,-1,-1,-1,-1,    /* 010 - 017 illegal */
115:     -1,-1,-1,-1,-1,-1,-1,-1,    /* 020 - 027 illegal */
116:     -1,-1,-1,-1,-1,-1,-1,-1,    /* 030 - 037 illegal */
117:
118:     0,          /* 040 space */
119:     42,         /* 041 ! */
120:     -1,         /* 042 " illegal */
121:     11,         /* 043 # */
122:     43,         /* 044 $ */
123:     28,         /* 045 % also ( */
124:     48,         /* 046 & also + */
125:     12,         /* 047 ' also @ */
126:
127:     28,         /* 050 ( also % */
128:     60,         /* 051 ) also lozenge */
129:     44,         /* 052 * */
130:     48,         /* 053 + also & */
131:     27,         /* 055 , */
132:     32,         /* 055 - */

```

```
133:      59,          /* 056 . */
134:      17,          /* 057 / */
135:
136:      10,          /* 060 0 */
137:      1,           /* 061 1 */
138:      2,           /* 062 2 */
139:      3,           /* 063 3 */
140:      4,           /* 064 4 */
141:      5,           /* 065 5 */
142:      6,           /* 066 6 */
143:      7,           /* 067 7 */
144:
145:      8,           /* 070 8 */
146:      9,           /* 071 9 */
147:      13,          /* 072 : */
148:      46,          /* 073 ; */
149:      62,          /* 074 < */
150:      11,          /* 075 = also # */
151:      14,          /* 076 > */
152:      58,          /* 077 ? */
153:
154:      12,          /* 0100 @ */
155:      49,          /* 0101 A */
156:      50,          /* 0102 B */
157:      51,          /* 0103 C */
158:      52,          /* 0104 D */
159:      53,          /* 0105 E */
160:      54,          /* 0106 F */
161:      55,          /* 0107 G */
162:
163:      56,          /* 0110 H */
164:      57,          /* 0111 I */
165:      33,          /* 0112 J */
166:      34,          /* 0113 K */
167:      35,          /* 0114 L */
168:      36,          /* 0115 M */
169:      37,          /* 0116 N */
170:      38,          /* 0117 O */
171:
172:      39,          /* 0120 P */
173:      40,          /* 0121 Q */
174:      41,          /* 0122 R */
175:      18,          /* 0123 S */
176:      19,          /* 0124 T */
177:      20,          /* 0125 U */
178:      21,          /* 0126 V */
179:      22,          /* 0127 W */
180:
181:      23,          /* 0130 X */
182:      24,          /* 0131 Y */
183:      25,          /* 0132 Z */
184:      61,          /* 0133 [ */
185:      30,          /* 0134 \ */
186:      45,          /* 0135 ] */
187:      29,          /* 0136 ^ word separator */
188:      -1,          /* 0137 _ illegal */
189:
190:      -1,          /* 0140 ` illegal */
191:      49,          /* 0141 a is A */
192:      16,          /* 0142 b is substitute blank */
193:      51,          /* 0143 c is C */
194:      52,          /* 0144 d is D */
195:      53,          /* 0145 e is E */
196:      54,          /* 0146 f is F */
197:      55,          /* 0147 g is G */
198:
```

```
199:    56,          /* 0150 h is H */
200:    57,          /* 0151 i is I */
201:    33,          /* 0152 j is J */
202:    34,          /* 0153 k is K */
203:    35,          /* 0154 l is L */
204:    36,          /* 0155 m is M */
205:    37,          /* 0156 n is N */
206:    38,          /* 0157 o is O */
207:
208:    39,          /* 0160 p is P */
209:    40,          /* 0161 q is Q */
210:    41,          /* 0162 r is R */
211:    18,          /* 0163 s is S */
212:    19,          /* 0164 t is T */
213:    20,          /* 0165 u is U */
214:    21,          /* 0166 v is V */
215:    22,          /* 0167 w is W */
216:
217:    23,          /* 0170 x is X */
218:    24,          /* 0171 y is Y */
219:    25,          /* 0172 z is Z */
220:    -1,          /* 0173 { illegal */
221:    26,          /* 0174 | substitute record mark */
222:    -1,          /* 0175 } illegal */
223:    -1,          /* 0176 ~ illegal */
224:    47,          /* 0177 □ delta */
225:
226:    -1,-1,-1,-1,-1,-1,-1,-1, /* 0200-0207 illegal */
227:    -1,-1,-1,-1,-1,-1,-1,-1, /* 0210-0217 illegal */
228:    -1,-1,-1,-1,-1,-1,-1,-1, /* 0220-0227 illegal */
229:    -1,-1,-1,-1,-1,-1,-1,-1, /* 0230-0237 illegal */
230:    -1,-1,-1,-1,-1,-1,-1,-1, /* 0240-0247 illegal */
231:    -1,-1,-1,-1,-1,-1,-1,-1, /* 0250-0257 illegal */
232:    -1,-1,-1,-1,-1,-1,-1,-1, /* 0260-0267 illegal */
233:    -1,-1,-1,-1,-1,-1,-1,-1, /* 0270-0277 illegal */
234:    -1,-1,-1,-1,-1,-1,-1,-1, /* 0300-0307 illegal */
235:
236:    -1,-1,-1,-1,-1,-1,      /* 0310-0315 illegal */
237:    63,          /* 0316 group mark */
238:    -1,          /* 0317 illegal */
239:
240:    -1,-1,-1,-1,-1,-1,-1,   /* 0320-0326 illegal */
241:    31,          /* 0327 segment mark */
242:
243:    26,          /* 0330 record mark */
244:    -1,-1,-1,-1,-1,-1,-1,   /* 0331-0337 illegal */
245:
246:    -1,-1,-1,-1,-1,-1,-1,-1, /* 0340-0347 illegal */
247:    -1,-1,-1,-1,-1,-1,-1,-1, /* 0350-0357 illegal */
248:    -1,-1,-1,-1,-1,-1,-1,-1, /* 0360-0367 illegal */
249:    -1,-1,-1,          /* 0370-0372 illegal */
250:    15,          /* 373 radical */
251:    -1,-1,-1,-1};        /* 0374-0377 illegal */
252:
253: //
254: // Parity table. Encode the 64 BCD characters. Does not itself
255: // include the check bit, so the 2nd 64 are the same as the first
256: // 64. Does include the WM bit, so the next 128 entries are the
257: // opposite of the first 128.
258: //
259:
260: int parity_table[256] = {
261: 0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,
262: 1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,0,1,1,0,0,1,0,1,1,0,0,1,
263:
264: 0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,
```

```

265:     1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,
266:
267:     1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,
268:     0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1,1,0,0,1,0,0,1,
269:
270:     1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,
271:     0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1,1,0,0,1,0,0,1
272: };
273:
274: // This table translates valide BCD numerice (odd parity, including
275: // check bit). It assumes that any WM bit has already been stripped.
276: // Note that the Two Out of Five code "0" bit is here coded as 16.
277:
278: int bcd_to_two_of_five_table[] = {
279:     0,17,18,0,20,0,0,12,24,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
280:     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
281:     0,0,0,3,0,5,6,0,0,9,71,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
282:     0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
283: };
284:
285: int two_of_five_to_bin_table[] = {
286:     -1,-1,-1,3,-1,5,6,-1,-1,9,0,-1,7,-1,-1,-1,
287:     -1,1,2,-1,4,-1,-1,-1,8,-1,-1,-1,-1,-1,-1
288: };
289:
290: int bin_to_two_of_five_table[] = {
291:     10,17,18,3,20,5,6,12,24,9
292: };
293:
294:

```

Parity - tabl. [256] does not include odd parity

Odd-parity-table [256]  
includes C, um  
1 if odd

```
1: //-----
2: #ifndef UI1410CPUH
3: #define UI1410CPUH
4:
5: #include "ubcd.h"
6:
7: extern long ten_thousands[], thousands[], hundreds[], tens[];
8: extern long scan_mod[];
9:
10: //
11: // Classes (types) used to implement the emulator, including the
12: // final class defining what is in the CPU, T1410CPU.
13:
14: //
15: // Abstract class designed to build lists of objects affected by Program
16: // Reset and Computer Reset
17: //
18:
19: class TCpuObject : public TObject {
20:
21: public:
22:     TCpuObject();                                // Constructor to init data
23:     virtual void OnComputerReset() = 0;           // Called during Computer Reset
24:     virtual void OnProgramReset() = 0;             // Called during Program Reset
25:
26: protected:
27:     bool DoesProgramReset;                      // true if this is reset by P.R. button
28:
29: public:
30:     TCpuObject *NextReset;
31: };
32:
33:
34: //
35: // A second abstract class of objects that not only react to the Resets,
36: // but also have entries on the display panel.
37: //
38:
39: class TDisplayObject : public TCpuObject {
40:
41: public:
42:     TDisplayObject();                            // Constructor to init data.
43:     virtual void Display() = 0;                  // Called to display this item
44:     virtual void LampTest(bool b) = 0;            // Called to start/end lamp test
45:
46: public:
47:     TDisplayObject *NextDisplay;
48: };
49:
50: □
51:
52: // Class TDisplayObjects are indicators: They just
53: // display other things. As a result, they need a pointer to
54: // a function returning bool in order to decide what to do.
55:
56: class TDisplayIndicator : public TDisplayObject {
57:
58: protected:
59:     TLabel *lamp;
60:     bool (_closure *display)();
61:
62: public:
63:
64:     // The constructor requires a pointer to a lamp and a pointer to
65:     // a function that can calculate lamp state.
66:
```

```
67:     // We use a closure so that we don't have to pass a pointer to the
68:     // stuff the display function needs (an object pointer is automatically
69:     // embedded in an __closure *)
70:
71:     TDisplayIndicator(TLabel *l,bool (__closure *func)() ) {
72:         lamp = l;
73:         display = func;
74:     }
75:
76:     virtual void OnComputerReset() { ; }      // These have no state to reset
77:     virtual void OnProgramReset() { ; }        // These have no state to reset
78:
79:     void Display() {
80:         lamp -> Enabled = display();
81:         lamp -> Repaint();
82:     }
83:
84:     void LampTest(bool b) {
85:         lamp -> Enabled = (b ? true : display());
86:         lamp -> Repaint();
87:     }
88: };
89:
90: □
91:
92: //  

93: // Class of TDisplayObjects that are latches:  

94: // that can be set, reset and their state read out. Some are  

95: // reset by Program Reset (PR) some are not.  

96: //  

97:
98: class TDisplayLatch : public TDisplayObject {
99:
100: protected:
101:     bool state;                                // Latches can be set or reset
102:     bool doprogramreset;                        // Some are reset by PR, some are not.
103:     TLabel *lamp;                             // Pointer to display lamp.
104:
105: public:
106:     TDisplayLatch(TLabel *l);                  // Constructor - Set up lamp
107:     TDisplayLatch(TLabel *l, bool progreset); // Same, but inhibit PR
108:
109:     virtual void OnComputerReset();           // Define Computer Reset behavior now
110:     virtual void OnProgramReset();            // Define Program Reset behavior now too
111:
112:     void Display();                          // Define display behavior
113:     void LampTest(bool b);                  // Define lamp test behavior.
114:
115:     // All you can really do with latches is set/reset/test them
116:
117:     inline void Reset() { state = false; }
118:     inline void Set() { state = true; }
119:     inline void Set(bool b) { state = b; }
120:     void SetStop();
121:     inline bool State() { return state; }
122: };
123:
124: □
125:
126: class TRingCounter : public TDisplayObject {
127: private:
128:     char state;                                // Override "state" variable !!
129:     char max;                                  // Max number of entries
130:     TLabel *lastlamp;                         // Last lamp to be displayed
131:
132: public:
```

```
133:     TLabel **lamps;           // Ptr to array of lamps
134:
135: public:
136:     TRingCounter(char n);      // Construct with # of entries
137:                                         // Ring counters are always reset by PR
138:
139:     virtual __fastcall ~TRingCounter(); // Destructor for array of lamps
140:
141: // Functions inherited from abstract base classes now need definition
142:
143: void OnComputerReset();
144: void OnProgramReset();
145: void Display();
146: void LampTest(bool b);
147:
148: // The real meat of the Ring Counter class
149:
150: inline void Reset() { state = 0; }
151: inline char Set(char n) {return state = n; }
152: inline char State() { return state; }
153: char Next();
154: };
155:
156: □
157:
158: // Data Registers. All are stored as BCD, but we have special
159: // set routines to set or clear special parts for those registers
160: // that don't use all the bits (e.g. the Op register has no WM or
161: // C bits.
162:
163: // Also, a Register can have an optional pointer to an error latch.
164: // If so, then whenever the register is used, a parity check for ODD
165: // parity is made, and if invalid, the error latch is set. (This
166: // includes use during assignment).
167:
168: class TRegister : public TCpuObject {
169:
170: private:
171:
172:     BCD value;
173:
174: public:
175:
176:     TRegister() { value = BITC; DoesProgramReset = true; }
177:     TRegister(bool b) { value = BITC; DoesProgramReset = b; }
178:     TRegister(int i) { value = i; DoesProgramReset = true; }
179:     TRegister(int i, bool b) { value = i; DoesProgramReset = b; }
180:
181:     inline void OnComputerReset() { Reset(); }
182:     inline void Reset() { value = BITC; }
183:
184:     inline void Set(BCD bcd) { value = bcd; }
185:
186:     inline BCD Get() { return value; }
187:
188:     void operator=(TRegister &source);
189:
190:     void OnProgramReset() {
191:         if(DoesProgramReset) {
192:             Reset();
193:         }
194:     }
195: };
196: □
197: // Address Registers. For efficiency, we keep both binary and
198: // the real 2-out-of-5 code representations. If either one is
```

```
199: // valid, and the other representation is requested, we convert
200: // on the fly (and mark that representation valid).
201:
202: // The Set() function effectively implements the Address Channel.
203:
204: class TAddressRegister : public TCpuObject {
205:
206: private:
207:
208:     long i_value;           // Integer equivalent of register
209:     bool i_valid;          // True if integer rep. is valid
210:     bool set[5];           // True if corresponding digit is set
211:     TWOOF5 digits[5];      // Original 2 out of 5 code representation
212:     bool d_valid;          // True if digit rep. is valid
213:
214: public:
215:
216:     void OnComputerReset() { Reset(); };
217:     void OnProgramReset() { };
218:
219:     TAddressRegister();    // Constructor / initialization
220:     bool IsValid();        // Returns true if all digits set
221:     long Gate();           // Returns integer value if valid, -1 if not.
222:     BCD GateBCD(int i);   // Returns a single digit
223:     void Set(TWOOF5 digit,int index); // Sets a digit
224:     void Set(long value);  // Sets whole register from binary (address mod)
225:     void Reset();          // Resets the register to blanks
226:
227:     void operator=(TAddressRegister &source); // Assignment
228:
229: };
230:
231: □
232:
233: // This class defines what is in an I/O Channel
234:
235: #define IOCHNOTREADY 1
236: #define IOCHBUSY 2
237: #define IOCHDATACHECK 4
238: #define IOCHCONDITION 8
239: #define IOCHNOTTRANSFER 16
240: #define IOCHWLRECORD 32
241:
242: #define IOLAMPNOTREADY 0
243: #define IOLAMPBUSY 1
244: #define IOLAMPDATACHECK 2
245: #define IOLAMPCONDITION 3
246: #define IOLAMPNOTTRANSFER 4
247: #define IOLAMPWLRECORD 5
248:
249: class T1410Channel : public TDisplayObject {
250:
251: public:
252:
253:     // Functions inherited from abstract base class classes now need definition
254:
255:     void OnComputerReset();
256:     void OnProgramReset();
257:     void Display();
258:     void LampTest(bool b);
259:
260: private:
261:
262:     // Channel information
263:
264:     int ChStatus;           // Channel status (see defines)
```

D:\TEST\1410\UI1410CPUUT.h

```
265:
266:     TLabel *ChStatusDisplay[6];           // Channel status lights
267:
268: public:
269:
270:     TRegister *ChOp;
271:     TRegister *ChUnitType;
272:     TRegister *ChUnitNumber;
273:     TRegister *ChR1, *ChR2;
274:
275:     TDisplayLatch *ChInterlock;
276:     TDisplayLatch *ChRBCTerlock;
277:     TDisplayLatch *ChRead;
278:     TDisplayLatch *ChWrite;
279:     TDisplayLatch *ChOverlap;
280:     TDisplayLatch *ChNotOverlap;
281:
282:     enum TapeDensity {
283:         DENSITY_200_556 = 0, DENSITY_200_800 = 1, DENSITY_556_800 = 2
284:     } TapeDensity;
285:
286:
287: // Methods
288:
289: T1410Channel(                                     // Constructor
290:     TLabel *LampInterlock,
291:     TLabel *LampRBCTerlock,
292:     TLabel *LampRead,
293:     TLabel *LampWRIte,
294:     TLabel *LampOverlap,
295:     TLabel *LampNotOverlap,
296:     TLabel *LampNotRead,
297:     TLabel *LampBusy,
298:     TLabel *LampDataCheck,
299:     TLabel *LampCondition,
300:     TLabel *LampWLRecord,
301:     TLabel *LampNoTransfer
302: );
303:
304: inline int SetStatus(int i) { return ChStatus = i; }
305: inline int GetStatus() { return ChStatus; }
306: };
307:
308:
309: □
310: // This class defines what is actually inside the CPU.
311:
312: #define MAXCHANNEL 2
313: #define CHANNEL1 0
314: #define CHANNEL2 1
315:
316: #define STORAGE 80000
317:
318: #define I_RING_OP 0
319: #define I_RING_1 1
320: #define I_RING_2 2
321: #define I_RING_3 3
322: #define I_RING_4 4
323: #define I_RING_5 5
324: #define I_RING_6 6
325: #define I_RING_7 7
326: #define I_RING_8 8
327: #define I_RING_9 9
328: #define I_RING_10 10
329: #define I_RING_11 11
330: #define I_RING_12 12
```

D:\TEST\1410\UI1410CPU.T.h

```
331:
332: #define A_RING_1 0
333: #define A_RING_2 1
334: #define A_RING_3 2
335: #define A_RING_4 3
336: #define A_RING_5 4
337: #define A_RING_6 5
338:
339: #define CLOCK_A 0
340: #define CLOCK_B 1
341: #define CLOCK_C 2
342: #define CLOCK_D 3
343: #define CLOCK_E 4
344: #define CLOCK_F 5
345: #define CLOCK_G 6
346: #define CLOCK_H 7
347: #define CLOCK_J 8
348: #define CLOCK_K 9
349:
350: #define SCAN_N 0
351: #define SCAN_1 1
352: #define SCAN_2 2
353: #define SCAN_3 3
354:
355: #define SUB_SCAN_NONE 0
356: #define SUB_SCAN_U 1
357: #define SUB_SCAN_B 2
358: #define SUB_SCAN_E 3
359: #define SUB_SCAN_MQ 4
360:
361: #define CYCLE_A 0
362: #define CYCLE_B 1
363: #define CYCLE_C 2
364: #define CYCLE_D 3
365: #define CYCLE_E 4
366: #define CYCLE_F 5
367: #define CYCLE_I 6
368: #define CYCLE_X 7
369:
370: class T1410CPU {
371:
372: private:
373:
374:     BCD core[STORAGE];
375:
376: public:
377:
378:     // Wiring list
379:
380:     TCpuObject *ResetList;           // List of latches.
381:     TDisplayObject *DisplayList;    // List of displayable things
382:
383:     // Data Registers
384:
385:     TRegister *A_Reg, *B_Reg, *Op_Reg, *Op_Mod_Reg;
386:
387:     // Address Registers
388:
389:     TAddressRegister *STAR;        // Storage Address Register
390:                                         // AKA MAR (Memory Address Register)
391:
392:     TAddressRegister *A_AR, *B_AR, *C_AR, *D_AR, *E_AR, *F_AR;
393:     TAddressRegister *I_AR;
394:
395:
396:     // Channels
```

D:\TEST\1410\UI1410CPUUT.h

```

397:     T1410Channel *Channel[MAXCHANNEL];           // 2 I/O Channels.
398:
399:
400:     // Indicators
401:     TDisplayIndicator *OffNormal;               // OFF NORMAL Indicator
402:
403:
404:     // Ring Counters
405:
406:     TRingCounter *IRing;                      // Instruction decode ring
407:     TRingCounter *ARing;                      // Address decode ring
408:     TRingCounter *ClockRing;                  // Cycle Clock
409:     TRingCounter *ScanRing;                   // Address Modification Mode
410:     TRingCounter *SubScanRing;                // Arithmetic Scan type
411:     TRingCounter *CycleRing;                  // CPU Cycle type
412:
413:     // Latches
414:
415:
416:     TDisplayLatch *CarryIn;                   // Carry latch
417:     TDisplayLatch *CarryOut;                  // Adder has generated carry
418:     TDisplayLatch *AComplement;              // A channel complement
419:     TDisplayLatch *BComplement;              // B channel complement
420:     TDisplayLatch *CompareBGTA;             // B > A
421:     TDisplayLatch *CompareBEQA;             // B = A
422:     TDisplayLatch *CompareBLTA;             // B < A NOTE: On after C. Reset.
423:     TDisplayLatch *Overflow;                 // Arithmetic Overflow
424:     TDisplayLatch *DivideOverflow;          // Divide Overflow
425:     TDisplayLatch *ZeroBalance;              // Zero arithmetic result
426:
427:     // Check Latches
428:
429:     TDisplayLatch *AChannelCheck;            // A Channel parity error
430:     TDisplayLatch *BChannelCheck;            // B Channel parity error
431:     TDisplayLatch *AssemblyChannelCheck;    // Assembly Channel parity error
432:     TDisplayLatch *AddressChannelCheck;     // Address Channel parity error
433:     TDisplayLatch *AddressExitCheck;        // Validity error at address reg.
434:     TDisplayLatch *ARegisterSetCheck;       // A register failed to reset
435:     TDisplayLatch *BRegisterSetCheck;       // B register failed to reset
436:     TDisplayLatch *OpRegisterSetCheck;      // Op register failed to set
437:     TDisplayLatch *OpModifierSetCheck;      // Op modifier failed to set
438:     TDisplayLatch *ACharacterSelectCheck;   // Incorrect A channel gating
439:     TDisplayLatch *BCharacterSelectCheck;   // Incorrect B channel getting
440:
441:     TDisplayLatch *IOInterlockCheck;        // Program did not check I/O
442:     TDisplayLatch *AddressCheck;            // Program gave bad address
443:     TDisplayLatch *RBCInterlockCheck;       // Program did not check RBC
444:     TDisplayLatch *InstructionCheck;        // Program issued invalide op
445:
446:     // Switches
447:
448:     enum Mode {                                // Mode switch, values must match
449:         MODE_RUN = 0, MODE_DISPLAY = 1, MODE_ALTER = 2,
450:         MODE_CE = 3, MODE_IE = 4, MODE_ADDR = 5
451:     } Mode;
452:
453:     enum AddressEntry {
454:         ADDR_ENTRY_I = 0, ADDR_ENTRY_A = 1, ADDR_ENTRY_B = 2,
455:         ADDR_ENTRY_C = 3, ADDR_ENTRY_D = 4, ADDR_ENTRY_E = 5,
456:         ADDR_ENTRY_F = 6
457:     } AddressEntry;
458:
459:     // The entry below is for the state of the 1415 Storage Scan SWITCH
460:     // (The storage scan modification mode is in the Ring ScanRing)
461:
462:     enum StorageScan {

```

D:\TEST\1410\UI1410CPUUT.h

```
463:     SSCAN_OFF = 0, SSCAN_LOAD_1 = 1, SSCAN_LOAD_0 = 2,
464:     SSCAN_REGEN_0 = 3, SSCAN_REGEN_1 = 4
465: } StorageScan;
466:
467: enum CycleControl {
468:     CYCLE_OFF = 0, CYCLE_LOGIC = 1, CYCLE_STORAGE = 2
469: } CycleControl;
470:
471: enum CheckControl {
472:     CHECK_STOP = 0, CHECK_RESTART = 1, CHECK_RESET = 2
473: } CheckControl;
474:
475: bool DiskWrInhibit;
476: bool AsteriskInsert;
477: bool InhibitPrintOut;
478:
479: bool StopLatch;
480: bool StopKeyLatch;
481: bool DisplayModeLatch;
482:
483: BCD BitSwitches;
484:
485: // Methods
486:
487: T1410CPU();                                // Constructor
488: void Display();                            // Run thru the display list
489:
490: private:
491:
492: // Indicator Routines
493: // Normally, these will be accessed via a bool (__closure *func)()
494:
495: bool IndicatorOffNormal();
496:
497: public:
498:
499: // Dynamic Channels
500:
501: static BCD AssemblyChannel();
502:
503: public:
504:
505: // Core operations (using STAR/MAR and B Data Register)
506:
507: void Readout();                           // Reads out one storage character
508: void Store();                            // Stores character in B Data Register
509: void Store(BCD bcd);                   // Sets B Data Register, then stores
510: void SetScan(char s);                  // Sets Scan Modification value
511:
512: long STARScan();                        // Applies Scan modification to STAR,
513:                                         // and returns the results - suitable to assign
514:
515: };
516:
517: extern T1410CPU *CPU;
518:
519: //-----
520: #endif
521:
```

```
1: //-----
2: #ifndef UI1410CPUH
3: #define UI1410CPUH
4:
5: #include "ubcd.h"
6:
7: //
8: // Classes (types) used to implement the emulator, including the
9: // final class defining what is in the CPU, T1410CPU.
10:
11: //
12: // Abstract class designed to build lists of objects affected by Program
13: // Reset and Computer Reset
14: //
15:
16: class TCpuObject : public TObject {
17:
18: public:
19:     TCpuObject();                                // Constructor to init data
20:     virtual void OnComputerReset() = 0;           // Called during Computer Reset
21:     virtual void OnProgramReset() = 0;             // Called during Program Reset
22:
23: protected:
24:     bool DoesProgramReset;                      // true if this is reset by P.R. button
25:
26: public:
27:     TCpuObject *NextReset;
28: };
29:
30:
31: //
32: // A second abstract class of objects that not only react to the Resets,
33: // but also have entries on the display panel.
34: //
35:
36: class TDisplayObject : public TCpuObject {
37:
38: public:
39:     TDisplayObject();                            // Constructor to init data.
40:     virtual void Display() = 0;                  // Called to display this item
41:     virtual void LampTest(bool b) = 0;            // Called to start/end lamp test
42:
43: public:
44:     TDisplayObject *NextDisplay;
45: };
46:
47: □
48:
49: // Class TDisplayObjects are indicators: They just
50: // display other things. As a result, they need a pointer to
51: // a function returning bool in order to decide what to do.
52:
53: class TDisplayIndicator : public TDisplayObject {
54:
55: protected:
56:     TLabel *lamp;
57:     bool (*display)();
58:
59: public:
60:
61:     // The constructor requires a pointer to a lamp and a pointer to
62:     // a function that can calculate lamp state.
63:
64:     TDisplayIndicator(TLabel *l,bool (*func)() ) {
65:         lamp = l;
66:         display = func;
```

```
67:     }
68:
69:     virtual void OnComputerReset() { ; }      // These have no state to reset
70:     virtual void OnProgramReset() { ; }        // These have no state to reset
71:
72:     void Display() {
73:         lamp -> Enabled = display();
74:         lamp -> Repaint();
75:     }
76:
77:     void LampTest(bool b) {
78:         lamp -> Enabled = (b ? true : display());
79:         lamp -> Repaint();
80:     }
81: };
82:
83: □
84:
85: /**
86:  * Class of TDisplayObjects that are latches:
87:  * that can be set, reset and their state read out. Some are
88:  * reset by Program Reset (PR) some are not.
89: */
90:
91: class TDisplayLatch : public TDisplayObject {
92:
93: protected:
94:     bool state;                                // Latches can be set or reset
95:     bool doprogramreset;                      // Some are reset by PR, some are not.
96:     TLabel *lamp;                            // Pointer to display lamp.
97:
98: public:
99:     TDisplayLatch(TLabel *l);                // Constructor - Set up lamp
100:    TDisplayLatch(TLabel *l, bool progreset); // Same, but inhibit PR
101:
102:    virtual void OnComputerReset();          // Define Computer Reset behavior now
103:    virtual void OnProgramReset();           // Define Program Reset behavior now too
104:
105:    void Display();                         // Define display behavior
106:    void LampTest(bool b);                  // Define lamp test behavior.
107:
108:    // All you can really do with latches is set/reset/test them
109:
110:    inline void Reset() { state = false; }
111:    inline void Set() { state = true; }
112:    inline void Set(bool b) { state = b; }
113:    inline bool State() { return state; }
114: };
115:
116: □
117:
118: class TRingCounter : public TDisplayObject {
119: private:
120:     char state;                                // Override "state" variable !!
121:     char max;                                  // Max number of entries
122:     TLabel *lastlamp;                        // Last lamp to be displayed
123:
124: public:
125:     TLabel **lamps;                          // Ptr to array of lamps
126:
127: public:
128:     TRingCounter(char n);                    // Construct with # of entries
129:                                         // Ring counters are always reset by PR
130:
131:     virtual __fastcall ~TRingCounter(); // Destructor for array of lamps
132:
```

```
133:     // Functions inherited from abstract base classes now need definition
134:
135:     void OnComputerReset();
136:     void OnProgramReset();
137:     void Display();
138:     void LampTest(bool b);
139:
140:     // The real meat of the Ring Counter class
141:
142:     inline void Reset() { state = 0; }
143:     inline char Set(char n) { return state = n; }
144:     inline char State() { return state; }
145:     char Next();
146: };
147:
148: □
149:
150: // Data Registers. All are stored as BCD, but we have special
151: // set routines to set or clear special parts for those registers
152: // that don't use all the bits (e.g. the Op register has no WM or
153: // C bits.
154:
155: class TRegister : public TCpuObject {
156:
157: private:
158:
159:     BCD value;
160:
161: public:
162:
163:     TRegister() { value = BITC; DoesProgramReset = true; }
164:     TRegister(bool b) { value = BITC; DoesProgramReset = b; }
165:     TRegister(int i) { value = i; DoesProgramReset = true; }
166:     TRegister(int i,bool b) { value = i; DoesProgramReset = b; }
167:
168:     inline void OnComputerReset() { Reset(); }
169:     inline void Reset() { value = BITC; }
170:     inline void Set(BCD bcd) { value = bcd; }
171:     inline BCD Get() { return value; }
172:
173:     void OnProgramReset() {
174:         if(DoesProgramReset) {
175:             Reset();
176:         }
177:     }
178: };
179: □
180: // Address Registers. For efficiency, we keep both binary and
181: // the real 2-out-of-5 code representations. If either one is
182: // valid, and the other representation is requested, we convert
183: // on the fly (and mark that representation valid).
184:
185: class TAddressRegister : public TCpuObject {
186:
187: private:
188:
189:     int i_value;           // Integer equivalent of register
190:     bool i_valid;          // True if integer rep. is valid
191:     bool set[5];           // True if corresponding digit is set
192:     TWOOF5 digits[5];      // Original 2 out of 5 code representation
193:     bool d_valid;          // True if digit rep. is valid
194:
195: public:
196:
197:     void OnComputerReset() { Reset(); };
198:     void OnProgramReset() { };
```

D:\TEST\1410\UI1410CPUUT.h

```
199:
200:     TAddressRegister();      // Constructor / initialization
201:     bool IsValid();          // Returns true if all digits set
202:     int Gate();              // Returns integer value if valid, -1 if not.
203:     BCD GateBCD(i);          // Returns a single digit
204:     void Set(TWOOF5 digit,int index); // Sets a digit
205:     void Set(int value);     // Sets whole register from binary (address mod)
206:     void Reset();            // Resets the register to blanks
207:
208: };
209:
210: □
211:
212: // This class defines what is in an I/O Channel
213:
214: #define IOCHNOTREADY    1
215: #define IOCHBUSY        2
216: #define IOCHDATACHECK   4
217: #define IOCHCONDITION   8
218: #define IOCHNOTTRANSFER 16
219: #define IOCHWLRECORD   32
220:
221: #define IOLAMPNOTREADY  0
222: #define IOLAMPBUSY      1
223: #define IOLAMPDATACHECK 2
224: #define IOLAMPCONDITION 3
225: #define IOLAMPNOTTRANSFER 4
226: #define IOLAMPWLRECORD  5
227:
228: class T1410Channel : public TDisplayObject {
229:
230: public:
231:
232:     // Functions inherited from abstract base class classes now need definition
233:
234:     void OnComputerReset();
235:     void OnProgramReset();
236:     void Display();
237:     void LampTest(bool b);
238:
239: private:
240:
241:     // Channel information
242:
243:     int ChStatus;                  // Channel status (see defines)
244:
245:     TLabel *ChStatusDisplay[6];    // Channel status lights
246:
247: public:
248:
249:     TRegister *ChOp;
250:     TRegister *ChUnitType;
251:     TRegister *ChUnitNumber;
252:     TRegister *ChR1, *ChR2;
253:
254:     TDisplayLatch *ChInterlock;
255:     TDisplayLatch *ChRBCTInterlock;
256:     TDisplayLatch *ChRead;
257:     TDisplayLatch *ChWrite;
258:     TDisplayLatch *ChOverlap;
259:     TDisplayLatch *ChNotOverlap;
260:
261:     enum TapeDensity {
262:         DENSITY_200_556 = 0, DENSITY_200_800 = 1, DENSITY_556_800 = 2
263:     } TapeDensity;
264:
```

D:\TEST\1410\UI1410CPUUT.h

```
265:
266:     // Methods
267:
268:     T1410Channel(                                     // Constructor
269:         TLabel *LampInterlock,
270:         TLabel *LampRBCInterlock,
271:         TLabel *LampRead,
272:         TLabel *LampWRIte,
273:         TLabel *LampOverlap,
274:         TLabel *LampNotOverlap,
275:         TLabel *LampNotRead,
276:         TLabel *LampBusy,
277:         TLabel *LampDataCheck,
278:         TLabel *LampCondition,
279:         TLabel *LampWLRecord,
280:         TLabel *LampNoTransfer
281:     );
282:
283:     inline int SetStatus(int i) { return ChStatus = i; }
284:     inline int GetStatus() { return ChStatus; }
285: };
286:
287: □
288:
289: // This class defines what is actually inside the CPU.
290:
291: #define MAXCHANNEL 2
292: #define CHANNEL1 0
293: #define CHANNEL2 1
294:
295: #define I_RING_OP 0
296: #define I_RING_1 1
297: #define I_RING_2 2
298: #define I_RING_3 3
299: #define I_RING_4 4
300: #define I_RING_5 5
301: #define I_RING_6 6
302: #define I_RING_7 7
303: #define I_RING_8 8
304: #define I_RING_9 9
305: #define I_RING_10 10
306: #define I_RING_11 11
307: #define I_RING_12 12
308:
309: #define A_RING_1 0
310: #define A_RING_2 1
311: #define A_RING_3 2
312: #define A_RING_4 3
313: #define A_RING_5 4
314: #define A_RING_6 5
315:
316: #define CLOCK_A 0
317: #define CLOCK_B 1
318: #define CLOCK_C 2
319: #define CLOCK_D 3
320: #define CLOCK_E 4
321: #define CLOCK_F 5
322: #define CLOCK_G 6
323: #define CLOCK_H 7
324: #define CLOCK_J 8
325: #define CLOCK_K 9
326:
327: #define SCAN_N 0
328: #define SCAN_1 1
329: #define SCAN_2 2
330: #define SCAN_3 3
```

```
331:
332: #define SUB_SCAN_NONE 0
333: #define SUB_SCAN_U 1
334: #define SUB_SCAN_B 2
335: #define SUB_SCAN_E 3
336: #define SUB_SCAN_MQ 4
337:
338: #define CYCLE_A 0
339: #define CYCLE_B 1
340: #define CYCLE_C 2
341: #define CYCLE_D 3
342: #define CYCLE_E 4
343: #define CYCLE_F 5
344: #define CYCLE_I 6
345: #define CYCLE_X 7
346:
347: class T1410CPU {
348:
349: public:
350:
351:     // Wiring list
352:
353:     TCpuObject *ResetList;           // List of latches.
354:     TDisplayObject *DisplayList;    // List of displayable things
355:
356:     // Data Registers
357:
358:     TRegister *A_Reg, *B_Reg, *Op_Reg, *Op_Mod_Reg;
359:
360:     // Address Registers
361:
362:     TAddressRegister *STAR;        // Storage Address Register
363:                                         // AKA MAR (Memory Address Register)
364:
365:     TAddressRegister *A_AR, *B_AR, *C_AR, *D_AR, *E_AR, *F_AR;
366:     TAddressRegister *I_AR;
367:
368:
369:     // Channels
370:
371:     T1410Channel *Channel[MAXCHANNEL]; // 2 I/O Channels.
372:
373:     // Indicators
374:
375:     TDisplayIndicator *OffNormal;   // OFF NORMAL Indicator
376:
377:     // Ring Counters
378:
379:     TRingCounter *IRing;           // Instruction decode ring
380:     TRingCounter *ARing;           // Address decode ring
381:     TRingCounter *ClockRing;       // Cycle Clock
382:     TRingCounter *ScanRing;        // Address Modification Mode
383:     TRingCounter *SubScanRing;     // Arithmetic Scan type
384:     TRingCounter *CycleRing;       // CPU Cycle type
385:
386:     // Latches
387:
388:     TDisplayLatch *CarryIn;        // Carry latch
389:     TDisplayLatch *CarryOut;       // Adder has generated carry
390:     TDisplayLatch *AComplement;    // A channel complement
391:     TDisplayLatch *BComplement;    // B channel complement
392:     TDisplayLatch *CompareBGTA;   // B > A
393:     TDisplayLatch *CompareBEQA;   // B = A
394:     TDisplayLatch *CompareBLTA;   // B < A NOTE: On after C. Reset.
395:     TDisplayLatch *Overflow;      // Arithmetic Overflow
396:     TDisplayLatch *DivideOverflow; // Divide Overflow
```

D:\TEST\1410\UI1410CPU.h

```

397:     TDisplayLatch *ZeroBalance;           // Zero arithmetic result
398:
399:     // Check Latches
400:
401:     TDisplayLatch *AChannelCheck;        // A Channel parity error
402:     TDisplayLatch *BChannelCheck;        // B Channel parity error
403:     TDisplayLatch *AssemblyChannelCheck; // Assembly Channel parity error
404:     TDisplayLatch *AddressChannelCheck;  // Address Channel parity error
405:     TDisplayLatch *AddressExitCheck;     // Validity error at address reg.
406:     TDisplayLatch *ARegisterSetCheck;    // A register failed to reset
407:     TDisplayLatch *BRegisterSetCheck;    // B register failed to reset
408:     TDisplayLatch *OpRegisterSetCheck;   // Op register failed to set
409:     TDisplayLatch *OpModifierSetCheck;   // Op modifier failed to set
410:     TDisplayLatch *ACharacterSelectCheck; // Incorrect A channel gating
411:     TDisplayLatch *BCharacterSelectCheck; // Incorrect B channel geting
412:
413:     TDisplayLatch *IOInterlockCheck;      // Program did not check I/O
414:     TDisplayLatch *AddressCheck;         // Program gave bad address
415:     TDisplayLatch *RBCInterlockCheck;    // Program did not check RBC
416:     TDisplayLatch *InstructionCheck;    // Program issued invalide op
417:
418:     // Switches
419:
420:     enum Mode {                           // Mode switch, values must match
421:         MODE_RUN = 0, MODE_DISPLAY = 1, MODE_ALTER = 2,
422:         MODE_CE = 3, MODE_IE = 4, MODE_ADDR = 5
423:     } Mode;
424:
425:     enum AddressEntry {
426:         ADDR_ENTRY_I = 0, ADDR_ENTRY_A = 1, ADDR_ENTRY_B = 2,
427:         ADDR_ENTRY_C = 3, ADDR_ENTRY_D = 4, ADDR_ENTRY_E = 5,
428:         ADDR_ENTRY_F = 6
429:     } AddressEntry;
430:
431:     enum StorageScan {
432:         SSCAN_OFF = 0, SSCAN_LOAD_1 = 1, SSCAN_LOAD_0 = 2,
433:         SSCAN_REGEN_0 = 3, SSCAN_REGEN_1 = 4
434:     } StorageScan;
435:
436:     enum CycleControl {
437:         CYCLE_OFF = 0, CYCLE_LOGIC = 1, CYCLE_STORAGE = 2
438:     } CycleControl;
439:
440:     enum CheckControl {
441:         CHECK_STOP = 0, CHECK_RESTART = 1, CHECK_RESET = 2
442:     } CheckControl;
443:
444:     bool DiskWrInhibit;
445:     bool AsteriskInsert;
446:     bool InhibitPrintOut;
447:
448:     BCD BitSwitches;
449:
450:     // Methods
451:
452:     T1410CPU();                         // Constructor
453:     void Display();                    // Run thru the display list
454:
455: private:
456:
457:     // Indicator Routines
458:
459:     static bool IndicatorOffNormal();
460:
461: public:
462:
```

```
463:     // Dynamic Channels
464:
465:     static BCD AssemblyChannel();
466: }
467:
468: extern T1410CPU *CPU;
469:
470: //-----
471: #endif
472:
```

```
1: // This Unit provides the functionality behind some of my private
2: // types for the 1410 emulator
3:
4: -----
5: #include <vcl\vcl.h>
6: #include <assert.h>
7: #pragma hdrstop
8:
9: #include "UI1410CPUUT.h"
10: #include "UI1415L.h"
11: #include "UI1415CE.h"
12: #include "UI1410PWR.h"
13:
14: -----
15:
16: // Declarations for Borland VCL controls
17:
18: #include <vcl\Classes.hpp>
19: #include <vcl\Controls.hpp>
20: #include <vcl\StdCtrls.hpp>
21:
22: // Implementation of TCpuObject (Abstract Base Class)
23:
24: TCpuObject::TCpuObject()
25: {
26:     NextReset = CPU -> ResetList;
27:     CPU -> ResetList = this;
28: }
29:
30: // Implementation of TDisplayObject (Abstract Base Class)
31:
32: // What is special about these objects is that they are on the display list.
33:
34: TDisplayObject::TDisplayObject()
35: {
36:     NextDisplay = CPU -> DisplayList;
37:     CPU -> DisplayList = this;
38: }
39:
40: □
41:
42: // Implementation of TDisplayLatch (Display Latch Base Class)
43:
44: // The constructor initializes the latch and sets the pointer to a lamp.
45: // A pointer to a lamp is required.
46: // Default is to be reset by a Program Reset
47:
48: TDisplayLatch::TDisplayLatch(TLabel *l)
49: {
50:     state = false;
51:     doprogramreset = true;
52:     lamp = l;
53: }
54:
55: // The second constructor does the same thing, but is passed a variable
56: // which indicates whether or not the latch should be reset by Program
57: // Reset.
58:
59: TDisplayLatch::TDisplayLatch(TLabel *l, bool progreset)
60: {
61:     state = false;
62:     doprogramreset = progreset;
63:     lamp = l;
64: }
65:
66: // All Displayable Latches are reset on a COMPUTER RESET
```

```
67:
68: void TDisplayLatch::OnComputerReset()
69: {
70:     Reset();
71: }
72:
73: // Whether or not a displayable latch is reset by program reset depends
74: // on the latch.
75:
76: void TDisplayLatch::OnProgramReset()
77: {
78:     if(doprogramreset) {
79:         Reset();
80:     }
81: }
82:
83: // When the Display routine is called, it sets or resets the lamp,
84: // depending on the current state.
85:
86: void TDisplayLatch::Display()
87: {
88:     lamp -> Enabled = state;
89:     lamp -> Repaint();
90: }
91:
92: // On a lamp test, light all the lamp.
93: // On reset of a lamp test, display the current state.
94:
95: void TDisplayLatch::LampTest(bool b)
96: {
97:     lamp -> Enabled = (b ? true : state);
98:     lamp -> Repaint();
99: }
100:
101: □
102:
103: // Implementation of TRingCounter (Ring Counter Class)
104:
105: // The constructor sets the max state of the ring, and resets the ring
106: // It also allocates an array of pointers to the lamps. The creator must
107: // fill in that array, however. Initially, the lamp pointers are empty,
108: // (which means no lamp is attached to that state).
109:
110: TRingCounter::TRingCounter(char n)
111: {
112:     int i;
113:
114:     state = 0;
115:     max = n;
116:     lastlamp = 0;
117:
118:     lamps = new TLabel*[n];
119:     for(i=0; i < n; ++i) {
120:         lamps[i] = 0;
121:     }
122: }
123:
124: // The destructor frees up the array of lamps. Probably will never use.
125:
126: __fastcall TRingCounter::~TRingCounter()
127: {
128:     delete[] lamps;
129: }
130:
131: // All Ring counters are reset on PROGRAM and COMPUTER RESET
132:
```

```
133: void TRingCounter::OnComputerReset()
134: {
135:     state = 0;
136: }
137:
138: void TRingCounter::OnProgramReset()
139: {
140:     state = 0;
141: }
142:
143: // When the Display routine is called, it resets the lamp corresponding
144: // to the state when it last displayed, and then displays the current state
145: // If there is no lamp associated with a state (lamp pointer is null),
146: // then don't display any lamp.
147:
148: void TRingCounter::Display()
149: {
150:     if(lastlamp) {
151:         lastlamp -> Enabled = false;
152:         lastlamp -> Repaint();
153:     }
154:     if(lamps[state] == 0) {
155:         lastlamp = 0;
156:         return;
157:     }
158:     lamps[state] -> Enabled = true;
159:     lamps[state] -> Repaint();
160:     lastlamp = lamps[state];
161: }
162:
163: // On a lamp test, light all the associated lamps.
164: // On reset of a lamp test, clear them all, then display the current state.
165:
166: void TRingCounter::LampTest(bool b)
167: {
168:     int i;
169:
170:     for(i = 0; i < max; ++i) {
171:         if(lamps[i] != 0) {
172:             lamps[i] -> Enabled = b;
173:             lamps[i] -> Repaint();
174:         }
175:     }
176:     if(!b) {
177:         lastlamp = 0;
178:         this -> Display();
179:     }
180: }
181:
182: // Next advances to the next state (or back to the start if appropriate)
183: // Returns current state. This is only useful for true Ring counters.
184:
185: char TRingCounter::Next()
186: {
187:     if(++state >= max) {
188:         state = 0;
189:     }
190:     return(state);
191: }
192:
193: □
194:
195: // Implementation of Address Registers
196:
197: TAddressRegister::TAddressRegister()
198: {
```

```
199:     i_valid = false;
200:     d_valid = false;
201:     set[0] = set[1] = set[2] = set[3] = set[4] = false;
202:     DoesProgramReset = false;
203: }
204:
205: bool TAddressRegister::IsValid()
206: {
207:     if(i_valid || d_valid ||
208:         (set[0] && set[1] && set[2] && set[3] && set[4]) ) {
209:         return(true);
210:     }
211:     return(false);
212: }
213:
214: void TAddressRegister::Reset()
215: {
216:     TWOOF5 zero;
217:
218:     i_valid = false;
219:     d_valid = false;
220:     set[0] = set[1] = set[2] = set[3] = set[4] = false;
221:     digits[0] = digits[1] = digits[2] = digits[3] = digits[4] = zero;
222: }
223:
224: int TAddressRegister::Gate()
225: {
226:     if(i_valid) {
227:         return(i_value);
228:     }
229:     else if(IsValid()) {
230:         i_value = 10000*digits[0].ToInt() + 1000*digits[1].ToInt() +
231:             100*digits[2].ToInt() + 10*digits[3].ToInt() + digits[4].ToInt();
232:         i_valid = true;
233:         d_valid = true;
234:         return(i_value);
235:     }
236:     else {
237:         CPU -> AddressExitCheck -> Set();           // Address exit error!
238:         return(-1);
239:     }
240: }
241:
242: BCD TAddressRegister::GateBCD(int i)
243: {
244:     if(d_valid) {
245:         return(digits[i].ToBCD());
246:     }
247:     else if(IsValid()) {
248:         digits[4] = i_value % 10;
249:         digits[3] = i_value % 100 - digits[4].ToInt();
250:         digits[2] = i_value % 1000 - (10*digits[3].ToInt()) - digits[4].ToInt();
251:         digits[1] = i_value % 10000 - (100*digits[2].ToInt()) -
252:             (10*digits[3].ToInt()) - digits[4].ToInt();
253:         digits[0] = i_value - (1000*digits[1].ToInt()) -
254:             (100*digits[2].ToInt()) - (10*digits[3].ToInt()) - digits[4].ToInt();
255:         d_valid = true;
256:         set[0] = set[1] = set[2] = set[3] = set[4] = true;
257:         return(digits[i].ToBCD());
258:     }
259:     else {
260:         CPU -> AddressExitCheck -> Set();
261:         return(BCD(0));
262:     }
263: }
```

```
265: void TAddressRegister::Set(TWOOF5 digit,int i)
266: {
267:     digits[i-1] = digit;
268:     set[i-1] = true;
269: }
270:
271: void TAddressRegister::Set(int i)
272: {
273:     d_valid = false;
274:     set[0] = set[1] = set[2] = set[3] = set[4] = false;
275:     i_valid = true;
276:     i_value = i;
277: }
278:
279: □
280:
281: // Implementation of I/O Channel Class
282:
283: // Constructor.  Initializes state
284:
285: T1410Channel::T1410Channel(
286:     TLabel *LampInterlock,
287:     TLabel *LampRBCInterlock,
288:     TLabel *LampRead,
289:     TLabel *LampWrite,
290:     TLabel *LampOverlap,
291:     TLabel *LampNotOverlap,
292:     TLabel *LampNotReady,
293:     TLabel *LampBusy,
294:     TLabel *LampDataCheck,
295:     TLabel *LampCondition,
296:     TLabel *LampWLRecord,
297:     TLabel *LampNoTransfer ) {
298:
299: ChStatus = 0;
300: TapeDensity = DENSITY_200_556;
301:
302: ChStatusDisplay[IOLAMPNOTREADY] = LampNotReady;
303: ChStatusDisplay[IOLAMPBUSY] = LampBusy;
304: ChStatusDisplay[IOLAMPDATACHECK] = LampDataCheck;
305: ChStatusDisplay[IOLAMPCONDITION] = LampCondition;
306: ChStatusDisplay[IOLAMPNOTTRANSFER] = LampNoTransfer;
307: ChStatusDisplay[IOLAMPWLRECORD] = LampWLRecord;
308:
309: // Generally, the channel latches are *not* reset by Program Reset
310:
311: ChInterlock = new TDisplayLatch(LampInterlock,false);
312: ChRBCInterlock = new TDisplayLatch(LampRBCInterlock,false);
313: ChRead = new TDisplayLatch(LampRead,false);
314: ChWrite = new TDisplayLatch(LampWrite,false);
315: ChOverlap = new TDisplayLatch(LampOverlap,false);
316: ChNotOverlap = new TDisplayLatch(LampNotOverlap,false);
317:
318: // Generally, the channel registers are *not* reset by Program Reset
319:
320: ChOp = new TRegister(false);
321: ChUnitType = new TRegister(false);
322: ChUnitNumber = new TRegister(false);
323: ChR1 = new TRegister(false);
324: ChR2 = new TRegister(false);
325: }
326:
327: // Channel is reset during ComputerReset
328:
329: void T1410Channel::OnComputerReset()
330: {
```

```
331:     ChStatus = 0;
332:
333:     // Note: The objects which are TDisplayLatch objects will reset themselves
334: }
335:
336: // Channel is not reset during Program Reset
337:
338: void T1410Channel::OnProgramReset()
339: {
340:     // Channel not affected by Program Reset
341: }
342:
343: // Display Routine.
344:
345: void T1410Channel::Display() {
346:
347:     int i;
348:
349:     ChStatusDisplay[IOLAMPNOTREADY] -> Enabled =
350:         ((ChStatus & IOCHNOTREADY) != 0);
351:     ChStatusDisplay[IOLAMPBUSY] -> Enabled =
352:         ((ChStatus & IOCHBUSY) != 0);
353:     ChStatusDisplay[IOLAMPDATACHECK] -> Enabled =
354:         ((ChStatus & IOCHDATACHECK) != 0);
355:     ChStatusDisplay[IOLAMPCONDITION] -> Enabled =
356:         ((ChStatus & IOCHCONDITION) != 0);
357:     ChStatusDisplay[IOLAMPWLRECORD] -> Enabled =
358:         ((ChStatus & IOCHWLRECORD) != 0);
359:     ChStatusDisplay[IOLAMPNOTTRANSFER] -> Enabled =
360:         ((ChStatus & IOCHNOTTRANSFER) != 0);
361:
362:     for(i=0; i <= 5; ++i) {
363:         ChStatusDisplay[i] -> Repaint();
364:     }
365:
366:     // Although in most instances the following would be redundant,
367:     // because these objects are also on the CPU display list, we include
368:     // them here in case we want to display a channel separately.
369:
370:     ChInterlock -> Display();
371:     ChRBCTInterlock -> Display();
372:     ChRead -> Display();
373:     ChWrite -> Display();
374:     ChOverlap -> Display();
375:     ChNotOverlap -> Display();
376: }
377:
378: // Channel Lamp Test
379:
380: void T1410Channel::LampTest(bool b)
381: {
382:     int i;
383:
384:     // Note, we don't have to do anything to the TDisplayLatch objects in
385:     // the channel for lamp test. They will take care of themselves on a
386:     // lamp test.
387:
388:     if(!b) {
389:         for(i=0; i <= 5; ++i) {
390:             ChStatusDisplay[i] -> Enabled = true;
391:             ChStatusDisplay[i] -> Repaint();
392:         }
393:     }
394:     else {
395:         Display();
396:     }
```

```
397: }
398:
399: □
400:
401: // CPU object constructor. Essentially this method "wires" the 1410.
402:
403: T1410CPU::T1410CPU()
404: {
405:     CPU = this;
406:
407:     // Clear out the lists
408:
409:     DisplayList = 0;
410:     ResetList = 0;
411:
412:     // Set switches to initial states
413:
414:     Mode = MODE_RUN;
415:     AddressEntry = ADDR_ENTRY_I;
416:     StorageScan = SSCAN_OFF;
417:     CycleControl = CYCLE_OFF;
418:     CheckControl = CHECK_STOP;
419:     DiskWrInhibit = false;
420:     AsteriskInsert = true;
421:     InhibitPrintOut = false;
422:     BitSwitches = BCD(0);
423:
424:     // Build the various displayable components of the CPU
425:
426:     IRing = new TRingCounter(13);
427:     assert(F1415L -> Light_I_OP != 0);
428:     IRing -> lamps[0] = F1415L -> Light_I_OP;
429:     IRing -> lamps[1] = F1415L -> Light_I_1;
430:     IRing -> lamps[2] = F1415L -> Light_I_2;
431:     IRing -> lamps[3] = F1415L -> Light_I_3;
432:     IRing -> lamps[4] = F1415L -> Light_I_4;
433:     IRing -> lamps[5] = F1415L -> Light_I_5;
434:     IRing -> lamps[6] = F1415L -> Light_I_6;
435:     IRing -> lamps[7] = F1415L -> Light_I_7;
436:     IRing -> lamps[8] = F1415L -> Light_I_8;
437:     IRing -> lamps[9] = F1415L -> Light_I_9;
438:     IRing -> lamps[10] = F1415L -> Light_I_10;
439:     IRing -> lamps[11] = F1415L -> Light_I_11;
440:     IRing -> lamps[12] = F1415L -> Light_I_12;
441:
442:     ARing = new TRingCounter(6);
443:     ARing -> lamps[0] = F1415L -> Light_A_1;
444:     ARing -> lamps[1] = F1415L -> Light_A_2;
445:     ARing -> lamps[2] = F1415L -> Light_A_3;
446:     ARing -> lamps[3] = F1415L -> Light_A_4;
447:     ARing -> lamps[4] = F1415L -> Light_A_5;
448:     ARing -> lamps[5] = F1415L -> Light_A_6;
449:
450:     ClockRing = new TRingCounter(10);
451:     ClockRing -> lamps[0] = F1415L -> Light_Clk_A;
452:     ClockRing -> lamps[1] = F1415L -> Light_Clk_B;
453:     ClockRing -> lamps[2] = F1415L -> Light_Clk_C;
454:     ClockRing -> lamps[3] = F1415L -> Light_Clk_D;
455:     ClockRing -> lamps[4] = F1415L -> Light_Clk_E;
456:     ClockRing -> lamps[5] = F1415L -> Light_Clk_F;
457:     ClockRing -> lamps[6] = F1415L -> Light_Clk_G;
458:     ClockRing -> lamps[7] = F1415L -> Light_Clk_H;
459:     ClockRing -> lamps[8] = F1415L -> Light_Clk_J;
460:     ClockRing -> lamps[9] = F1415L -> Light_Clk_K;
461:
462:     ScanRing = new TRingCounter(4);
```

D:\TEST\1410\UI1410CPUT.cpp

```
463:     ScanRing -> lamps[0] = F1415L -> Light_Scan_N;
464:     ScanRing -> lamps[1] = F1415L -> Light_Scan_1;
465:     ScanRing -> lamps[2] = F1415L -> Light_Scan_2;
466:     ScanRing -> lamps[3] = F1415L -> Light_Scan_3;
467:
468:     SubScanRing = new TRingCounter(5);
469:     // NOTE: State 0 is "OFF" - no flip flops set
470:     SubScanRing -> lamps[1] = F1415L -> Light_Sub_Scan_U;
471:     SubScanRing -> lamps[2] = F1415L -> Light_Sub_Scan_B;
472:     SubScanRing -> lamps[3] = F1415L -> Light_Sub_Scan_E;
473:     SubScanRing -> lamps[4] = F1415L -> Light_Sub_Scan_MQ;
474:
475:     CycleRing = new TRingCounter(8);
476:     CycleRing -> lamps[0] = F1415L -> Light_Cycle_A;
477:     CycleRing -> lamps[1] = F1415L -> Light_Cycle_B;
478:     CycleRing -> lamps[2] = F1415L -> Light_Cycle_C;
479:     CycleRing -> lamps[3] = F1415L -> Light_Cycle_D;
480:     CycleRing -> lamps[4] = F1415L -> Light_Cycle_E;
481:     CycleRing -> lamps[5] = F1415L -> Light_Cycle_F;
482:     CycleRing -> lamps[6] = F1415L -> Light_Cycle_I;
483:     CycleRing -> lamps[7] = F1415L -> Light_Cycle_X;
484:
485:     // Build the various latches. Most of these are not
486:     // reset during Program Reset
487:
488:     CarryIn = new TDisplayLatch(F1415L -> Light_Carry_In, false);
489:     CarryOut = new TDisplayLatch(F1415L -> Light_Carry_Out, false);
490:     AComplement = new TDisplayLatch(F1415L -> Light_A_Complement, false);
491:     BComplement = new TDisplayLatch(F1415L -> Light_B_Complement, false);
492:
493:     CompareBGTA = new TDisplayLatch(F1415L -> Light_B_GT_A, false);
494:     CompareBEQA = new TDisplayLatch(F1415L -> Light_B_EQ_A, false);
495:     CompareBLTA = new TDisplayLatch(F1415L -> Light_B_LT_A, false);
496:     Overflow = new TDisplayLatch(F1415L -> Light_Overflow, false);
497:     DivideOverflow = new TDisplayLatch(F1415L -> Light_Divide_Overflow, false);
498:     ZeroBalance = new TDisplayLatch(F1415L -> Light_Zero_Balance, false);
499:
500:     // Build the various check latches. Program Reset does reset these
501:
502:     AChannelCheck = new TDisplayLatch(F1415L -> Light_Check_AChannel);
503:     BChannelCheck = new TDisplayLatch(F1415L -> Light_Check_BChannel);
504:     AssemblyChannelCheck = new TDisplayLatch(F1415L ->
505:         Light_Check_AssemblyChannel);
506:     AddressChannelCheck = new TDisplayLatch(F1415L ->
507:         Light_Check_AddressChannel);
508:     AddressExitCheck = new TDisplayLatch(F1415L -> Light_Check_AddressExit);
509:     ARegisterSetCheck = new TDisplayLatch(F1415L -> Light_Check_ARegisterSet);
510:     BRegisterSetCheck = new TDisplayLatch(F1415L -> Light_Check_BRegisterSet);
511:     OpRegisterSetCheck = new TDisplayLatch(F1415L -> Light_Check_OpRegisterSet);
512:     OpModifierSetCheck = new TDisplayLatch(F1415L -> Light_Check_OpModifierSet);
513:     ACharacterSelectCheck = new TDisplayLatch(F1415L ->
514:         Light_Check_ACharacterSelect);
515:     BCharacterSelectCheck = new TDisplayLatch(F1415L ->
516:         Light_Check_BCharacterSelect);
517:
518:     IOInterlockCheck = new TDisplayLatch(F1415L -> Light_Check_IOInterlock);
519:     AddressCheck = new TDisplayLatch(F1415L -> Light_Check_AddressCheck);
520:     RBCInterlockCheck = new TDisplayLatch(F1415L -> Light_Check_RBCInterlock);
521:     InstructionCheck = new TDisplayLatch(F1415L -> Light_Check_InstructionCheck);
522:
523:     // Build the Data Registers
524:     A_Reg = new TRegister();
525:     B_Reg = new TRegister();
526:     Op_Reg = new TRegister(0);
527:     Op_Mod_Reg = new TRegister(0);
```

```
525:  
526: // Build the Address Registers  
527:  
528: STAR = new TAddressRegister();  
529: A_AR = new TAddressRegister();  
530: B_AR = new TAddressRegister();  
531: C_AR = new TAddressRegister();  
532: D_AR = new TAddressRegister();  
533: E_AR = new TAddressRegister();  
534: F_AR = new TAddressRegister();  
535: I_AR = new TAddressRegister();  
536:  
537: // Build the channels  
538:  
539: Channel[CHANNEL1] = new T1410Channel(  
540:     F1415L -> Light_Ch1_Interlock,  
541:     F1415L -> Light_Ch1_RBCInterlock,  
542:     F1415L -> Light_Ch1_Read,  
543:     F1415L -> Light_Ch1_Write,  
544:     F1415L -> Light_Ch1_Overlap,  
545:     F1415L -> Light_Ch1_NoOverlap,  
546:     F1415L -> Light_Ch1_NotReady,  
547:     F1415L -> Light_Ch1_Busy,  
548:     F1415L -> Light_Ch1_DataCheck,  
549:     F1415L -> Light_Ch1_Condition,  
550:     F1415L -> Light_Ch1_WLRecord,  
551:     F1415L -> Light_Ch1_NoTransfer  
552: );  
553:  
554: Channel[CHANNEL2] = new T1410Channel(  
555:     F1415L -> Light_Ch2_Interlock,  
556:     F1415L -> Light_Ch2_RBCInterlock,  
557:     F1415L -> Light_Ch2_Read,  
558:     F1415L -> Light_Ch2_Write,  
559:     F1415L -> Light_Ch2_Overlap,  
560:     F1415L -> Light_Ch2_NoOverlap,  
561:     F1415L -> Light_Ch2_NotReady,  
562:     F1415L -> Light_Ch2_Busy,  
563:     F1415L -> Light_Ch2_DataCheck,  
564:     F1415L -> Light_Ch2_Condition,  
565:     F1415L -> Light_Ch2_WLRecord,  
566:     F1415L -> Light_Ch2_NoTransfer  
567: );  
568:  
569: // Some latches are set after power on...  
570:  
571: CompareBLTA -> Set();  
572: I_AR -> Set(1);  
573:  
574: // Finally, set up the indicators  
575:  
576: OffNormal = new TDisplayIndicator(F1415L -> Light_Off_Normal,  
577:                                     T1410CPU::IndicatorOffNormal);  
578:  
579: }  
580:  
581:  
582: // CPU Object display - run thru the display list  
583:  
584: void T1410CPU::Display()  
585: {  
586:     TDisplayObject *l;  
587:  
588:     for(l = DisplayList; l != 0; l = l -> NextDisplay) {  
589:         l -> Display();  
590:     }
```

```
591: }
592:
593: // Off Normal Indicator routine
594:
595: bool T1410CPU::IndicatorOffNormal()
596: {
597:     return(CPU -> InhibitPrintOut ||
598:            !(CPU -> AsteriskInsert) ||
599:            CPU -> CycleControl != CPU -> CYCLE_OFF ||
600:            CPU -> CheckControl != CPU -> CHECK_STOP ||
601:            (CPU -> Mode == CPU -> MODE_CE &&
602:             CPU -> StorageScan != CPU -> SSCAN_OFF) ||
603:            CPU -> AddressEntry != CPU -> ADDR_ENTRY_I );
604: }
605:
606: // Assembly Channel (Dummy, for now)
607:
608: BCD T1410CPU::AssemblyChannel()
609: {
610:     return(BITC);
611: }
```

```
1: //-----
2: #ifndef UI1415IOH
3: #define UI1415IOH
4: //-----
5: #include <vcl\Classes.hpp>
6: #include <vcl\Controls.hpp>
7: #include <vcl\StdCtrls.hpp>
8: #include <vcl\Forms.hpp>
9: //-----
10:
11: // 1415 Console User Interface
12:
13: #include "ubcd.h"
14: #include <vcl\ExtCtrls.hpp>
15:
16: #define CONSOLE_IDLE      1           // Console is idle
17: #define CONSOLE_NORMAL    2           // Normal Read Mode input
18: #define CONSOLE_LOAD      3           // Normal Load Mode input
19: #define CONSOLE.Alter     4           // Console is in alter mode loading memory
20: #define CONSOLE_ADDR      5           // Console is in address set or display mode
21:
22: #define CONSOLE_MATRIX_HOME 0        // Console matrix home position
23:
24: class TFI1415IO : public TForm
25: {
26:     __published: // IDE-managed Components
27:         TMemo *I1415IO;
28:         TLabel *KeyboardLock;
29:         TButton *InqReq;
30:         TButton *InqRise;
31:         TButton *InqCancel;
32:         TButton *WordMark;
33:         TButton *MarginRelease;
34:         TTimer *KeyboardLockReset;
35:         void __fastcall I1415IOKeyPress(TObject *Sender, char &Key);
36:
37:         void __fastcall I1415IOKeyDown(TObject *Sender, WORD &Key, TShiftState Shift);
38:
39:         void __fastcall WordMarkClick(TObject *Sender);
40:         void __fastcall MarginReleaseClick(TObject *Sender);
41:
42:         void __fastcall KeyboardLockResetTimer(TObject *Sender);
43: private: // User declarations
44:     int state;
45:     int matrix;
46:     void NextLine();
47:     void DoWordMark();
48:     void LockKeyboard();
49:     void UnlockKeyboard();
50: public: // User declarations
51:     __fastcall TFI1415IO(TComponent* Owner);
52:     void SendBCDTo1415(BCD bcd);
53:     bool SetState(int s);
54:     inline void SetMatrix(int i) { matrix = i; }
55:     void SetMatrix(int x, int y) {matrix = 6*(x-1)+y; }
56:     inline int GetMatrix() { return matrix; }
57:     void StepMatrix() { ++matrix; }
58:     void ResetMatrix() { matrix = CONSOLE_MATRIX_HOME; }
59:     void DoMatrix();
60: };
61: //-----
62: extern TFI1415IO *FI1415IO;
63: //-----
64:
65: // Keyboard representations of some unusual BCD characters
66:
```

```
67: #define KBD_RADICAL          022
68: #define KBD_RECORD_MARK      'I'
69: #define KBD_ALT_BLANK         002
70: #define KBD_WORD_SEPARATOR    '^'
71: #define KBD_SEGMENT_MARK     023
72: #define KBD_DELTA              004
73: #define KBD_GROUP_MARK        007
74: #define KBD_WORD_MARK         0x1b
75:
76: #endif
```

```
1: // This Unit defines the behavior of the 1415 console typewriter and
2: // associated things
3:
4: //-----
5: #include <vcl\vcl.h>
6: #pragma hdrstop
7:
8: #include "UI1415IO.h"
9: #include "ubcd.h"
10: #include "UI1410DEBUG.h"
11: #include "UI1410CPUT.h"
12:
13: //-----
14: #pragma resource "* .dfm"
15: TFI1415IO *FI1415IO;
16: //-----
17: __fastcall TFI1415IO::TFI1415IO(TComponent* Owner)
18:     : TForm(Owner)
19: {
20:     state = CONSOLE_IDLE;
21: }
22: //-
23: void __fastcall TFI1415IO::I1415IOKeyPress(TObject *Sender, char &Key)
24: {
25:     BCD bcd_key;
26:
27:     switch(Key) {
28:     case KBD_RADICAL:
29:         bcd_key = BCD::BCDConvert(ASCII_RADICAL);
30:         break;
31:     case KBD_RECORD_MARK:
32:         bcd_key = BCD::BCDConvert(ASCII_RECORD_MARK);
33:         break;
34:     case KBD_ALT_BLANK:
35:         bcd_key = BCD::BCDConvert(ASCII_ALT_BLANK);
36:         break;
37:     case KBD_WORD_SEPARATOR:
38:         bcd_key = BCD::BCDConvert(ASCII_WORD_SEPARATOR);
39:         break;
40:     case KBD_SEGMENT_MARK:
41:         bcd_key = BCD::BCDConvert(ASCII_SEGMENT_MARK);
42:         break;
43:     case KBD_DELTA:
44:         bcd_key = BCD::BCDConvert(ASCII_DELTA);
45:         break;
46:     case KBD_GROUP_MARK:
47:         bcd_key = BCD::BCDConvert(ASCII_GROUP_MARK);
48:         break;
49:     case KBD_WORD_MARK:      // Escape == Wordmark Key
50:         DoWordMark();
51:         return;
52:     case 'b':
53:         bcd_key = BCD::BCDConvert('B');
54:         break;
55:     default:
56:         if(BCD::BCDCheck(Key) < 0) {           // Return if unmapped key.
57:             LockKeyboard();
58:             return;
59:         }
60:         bcd_key = BCD::BCDConvert(Key);
61:     }
62:
63:     // Decide what to do with key, depending on console state.
64:     // Note that the really special keys (Wordmark, above, and the
65:     // console inquiry buttons), are checked elsewhere. This test
66:     // is for "normal" BCD characters only.
```

```
67:         switch(state) {
68:             case CONSOLE_IDLE:
69:                 LockKeyboard();                                // Only INQ Request allowed
70:                 return;
71:             case CONSOLE_NORMAL:
72:             case CONSOLE_LOAD:
73:             case CONSOLE_ALTER:
74:                 UnlockKeyboard();
75:                 SendBCDTo1415(bcd_key);
76:                 break;
77:             case CONSOLE_ADDR:
78:                 if(isdigit(Key)) {
79:                     UnlockKeyboard();
80:                     SendBCDTo1415(bcd_key);
81:                 }
82:                 else {
83:                     LockKeyboard();
84:                 }
85:                 break;
86:             }
87:         }
88:     }
89: //-----
90: void __fastcall TFI1415IO::I1415IOKeyDown(TObject *Sender, WORD &Key,
91:                                         TShiftState Shift)
92: {
93:
94:     // Special key handling: Treat Page Down as a Margin Release.
95:
96:     if(Key == VK_NEXT) {
97:         NextLine();
98:     }
99: }
100: //-----
101:
102: // Method to set the state of the console
103:
104: bool TFI1415IO::SetState(int s)
105: {
106:     switch(s) {
107:         case CONSOLE_IDLE:
108:             InqReq -> Enabled = true;
109:             InqRlse -> Enabled = false;
110:             InqCancel -> Enabled = false;
111:             WordMark -> Enabled = false;
112:             break;
113:         case CONSOLE_NORMAL:
114:             InqReq -> Enabled = true;
115:             InqRlse -> Enabled = true;
116:             InqCancel -> Enabled = true;
117:             WordMark -> Enabled = false;
118:             break;
119:         case CONSOLE_LOAD:
120:             InqReq -> Enabled = true;
121:             InqRlse -> Enabled = true;
122:             InqCancel -> Enabled = true;
123:             WordMark -> Enabled = true;
124:             break;
125:         case CONSOLE_ALTER:
126:             InqReq -> Enabled = false;
127:             InqRlse -> Enabled = false;
128:             InqCancel -> Enabled = false;
129:             WordMark -> Enabled = true;
130:             break;
131:         case CONSOLE_ADDR:
132:             InqReq -> Enabled = false;
```

```
133:         InqRlse -> Enabled = false;
134:         InqCancel -> Enabled = false;
135:         WordMark -> Enabled = false;
136:         break;
137:     default:
138:         return(false);
139:     }
140:     state = s;
141:     return(true);
142: }
143:
144:
145: //
146: // Utility routine to go to next line on console
147: //
148:
149: void TFI1415IO::NextLine()
150: {
151:     I1415IO -> Lines -> Add("");      // Placeholder for wordmarks
152:     I1415IO -> Lines -> Add("");
153: }
154:
155: //
156: // Utility routines to lock/unlock keyboard - display/clear light.
157: //
158:
159: void TFI1415IO::LockKeyboard()
160: {
161:     KeyboardLock -> Enabled = true;
162:     KeyboardLockReset -> Enabled = true;
163: }
164:
165: void TFI1415IO::UnlockKeyboard()
166: {
167:     KeyboardLock -> Enabled = false;
168:     KeyboardLockReset -> Enabled = false;
169: }
170:
171: //
172: // Utility routine to handle wordmark key
173: //
174:
175: void TFI1415IO::DoWordMark()
176: {
177:     int last;
178:
179:     if(state != CONSOLE_LOAD && state != CONSOLE_ALTER) {
180:         LockKeyboard();
181:         return;
182:     }
183:
184:     last = I1415IO -> Lines -> Count - 1;
185:     if(last <= 0) {
186:         NextLine();
187:         last = I1415IO -> Lines -> Count -1;
188:     }
189:
190:     if(I1415IO -> Lines -> Strings[last-1].Length() <=
191:        I1415IO -> Lines -> Strings[last].Length() ) {
192:         I1415IO -> Lines -> Strings[last-1] =
193:             I1415IO -> Lines -> Strings[last-1] + "v";
194:         UnlockKeyboard();
195:     }
196:     else {
197:         LockKeyboard();
198:     }
```

```
199: }
200:
201: //
202: // Utility routine to send data to console from
203: //
204:
205: void TFI1415IO::SendBCDTol415(BCD bcd)
206: {
207:     int last;
208:
209:     last = I1415IO -> Lines -> Count - 1;
210:     if(last <= 0) {
211:         NextLine();
212:         last = I1415IO -> Lines -> Count -1;
213:     }
214:
215:     if(I1415IO -> Lines -> Strings[last-1].Length() <=
216:         I1415IO -> Lines -> Strings[last].Length()) {
217:         I1415IO -> Lines -> Strings[last-1] =
218:             I1415IO -> Lines -> Strings[last-1] + " ";
219:     }
220:     I1415IO -> Lines -> Strings[last] =
221:         I1415IO -> Lines -> Strings[last] + bcd.ToAscii();
222:     if(I1415IO -> Lines -> Strings[last].Length() > 80) {
223:         NextLine();
224:     }
225:     I1415IO -> Modified = true;
226: }
227:
228: void TFI1415IO::DoMatrix()
229: {
230:     if(matrix < 0 || matrix > 42) {
231:         DEBUG("Invalid Console Matrix Position: %d",matrix);
232:         return;
233:     }
234:
235:     // Console Matrix Processing
236:
237:     // Positions 0 (Home) thru 36 are used for display/output purposes
238:
239:     switch(matrix) {
240:
241:     case 0:
242:         // Home position: do nothing
243:         break;
244:
245:     case 1:
246:     case 2:
247:     case 3:
248:     case 4:
249:     case 5:
250:         SendBCDTol415(CPU -> I_AR -> GateBCD(matrix));
251:         break;
252:
253:     case 6:
254:     case 12:
255:     case 18:
256:     case 21:
257:     case 25:
258:     case 31:
259:     case 36:
260:         SendBCDTol415(BCD::BCDConvert(' '));
261:
262:
263:     case 7:
264:     case 8:
```

```
265:     case 9:
266:     case 10:
267:     case 11:
268:         SendBCDTo1415(CPU -> A_AR -> GateBCD(matrix-6));
269:         break;
270:
271:     case 13:
272:     case 14:
273:     case 15:
274:     case 16:
275:     case 17:
276:         SendBCDTo1415(CPU -> B_AR -> GateBCD(matrix-12));
277:         break;
278:
279:     case 19:
280:         SendBCDTo1415(CPU -> Op_Reg -> Get());
281:         break;
282:
283:     case 20:
284:         SendBCDTo1415(CPU -> Op_Mod_Reg -> Get());
285:         break;
286:
287:     case 22:
288:         SendBCDTo1415(CPU -> A_Reg -> Get());
289:         break;
290:
291:     case 23:
292:         SendBCDTo1415(CPU -> B_Reg -> Get());
293:         break;
294:
295:     case 24:
296:         SendBCDTo1415(CPU -> AssemblyChannel());
297:         break;
298:
299:     case 26:
300:         SendBCDTo1415(CPU -> Channel[CHANNEL1] -> ChUnitType -> Get());
301:         break;
302:
303:     case 27:
304:         SendBCDTo1415(CPU -> Channel[CHANNEL1] -> ChUnitNumber -> Get());
305:         break;
306:
307:     case 28:
308: #if MAXCHANNEL > 1
309:         SendBCDTo1415(CPU -> Channel[CHANNEL2] -> ChUnitType -> Get());
310: #endif
311:         break;
312:
313:     case 29:
314: #if MAXCHANNEL > 1
315:         SendBCDTo1415(CPU -> Channel[CHANNEL2] -> ChUnitNumber -> Get());
316: #endif
317:         break;
318:
319:     case 34:
320:         NextLine();
321:         break;
322:
323: // Positions 37 thru 42 are to accept an address (not implemented yet)
324:
325:     default:
326:         break;
327:     }
328: }
329:
330: void __fastcall TFI1415IO::WordMarkClick(TObject *Sender)
```

```
331: {
332:     DoWordMark();
333:     FocusControl(I1415IO);
334: }
335: //-----
336: void __fastcall TFI1415IO::MarginReleaseClick(TObject *Sender)
337: {
338:     NextLine();
339:     FocusControl(I1415IO);
340: }
341: //-----
342: void __fastcall TFI1415IO::KeyboardLockResetTimer(TObject *Sender)
343: {
344:     UnlockKeyboard();
345: }
346: //-----
```

```
1: //-----
2: #include <vcl\vcl.h>
3: #pragma hdrstop
4:
5: #include "UI1410PWR.h"
6: //-----
7: #pragma resource "* .dfm"
8:
9: #include "UI1410DEBUG.h"
10:
11: // These methods define the operation of the 1410 power panel
12:
13: TFI1410PWR *FI1410PWR;
14: //-----
15: __fastcall TFI1410PWR::TFI1410PWR(TComponent* Owner)
16:     : TForm(Owner)
17: {
18:     Mode -> ItemIndex = CPU -> MODE_RUN;
19:     Height = 522;
20:     Width = 327;
21: }
22: //-----
23: void __fastcall TFI1410PWR::EmergencyOffClick(TObject *Sender)
24: {
25:     FI14101 -> Close();
26: }
27: //-----
28: void __fastcall TFI1410PWR::ComputerResetClick(TObject *Sender)
29: {
30:     TCpuObject *o;
31:
32:     DEBUG("Computer Reset",0)
33:     for(o = CPU -> ResetList; o != 0; o = o -> NextReset) {
34:         o -> OnComputerReset();
35:     }
36:
37:     // Handle any special case latches (which are reset in the above loop)
38:
39:     CPU -> CompareBLTA -> Set();
40:     CPU -> I_AR -> Set(1);
41:
42:     CPU -> Display();
43: }
44: //-----
45: void __fastcall TFI1410PWR::ModeChange(TObject *Sender)
46: {
47:     if(Mode -> ItemIndex >= 0) {
48:         CPU -> Mode = Mode -> ItemIndex;
49:     }
50:     CPU -> OffNormal -> Display();
51:     DEBUG("Mode Switch set to %d",CPU -> Mode)
52: }
53: //-----
54: void __fastcall TFI1410PWR::ProgramResetClick(TObject *Sender)
55: {
56:     TCpuObject *o;
57:
58:     DEBUG("Program Reset",0)
59:     for(o = CPU -> ResetList; o != 0; o = o -> NextReset) {
60:         o -> OnProgramReset();
61:     }
62:
63:     // Handle any special cases
64:
65:     CPU -> I_AR -> Set(1);
66:
```

```
67:     CPU -> Display();  
68: }  
69: //-----  
70:
```

```
1: #ifndef UBCDH
2: #define UBCDH
3:
4: //
5: //  Definition of character representation
6: //
7:
8: #define BITWM 0x80
9: #define BIT1 1
10: #define BIT2 2
11: #define BIT4 4
12: #define BIT8 8
13: #define BITA 0x10
14: #define BITB 0x20
15: #define BITC 0x40
16:
17: extern char bcd_ascii[];
18: extern int ascii_bcd[];
19:
20: //
21: //  These are defined so that keyboard input may recognize them
22: //
23:
24: #define ASCII_RECORD_MARK    0174
25: #define ASCII_GROUP_MARK     0316
26: #define ASCII_SEGMENT_MARK   0327
27: #define ASCII_RADICAL         0373
28: #define ASCII_ALT_BLANK       'b'
29: #define ASCII_WORD_SEPARATOR  '^'
30: #define ASCII_DELTA           0177
31:
32: class BCD {
33:
34: private:
35:     int c;                                // WM_B A 8 4 2 1
36:                                         // (Check bit is not used)
37: public:
38:     BCD() { c = 0; }                      // Default constructor
39:
40:     BCD(int i) { c = i; }
41:
42:     inline int BCDConvert(int ch) { // Constructor
43:         if(ascii_bcd[ch] < 0) {
44:             return ascii_bcd[ASCII_ALT_BLANK];
45:         }
46:         else {
47:             return ascii_bcd[ch];
48:         }
49:     }
50:
51:     inline int BCDCheck(int ch) {
52:         return(ascii_bcd[ch]);
53:     }
54:
55:     inline char ToAscii() {
56:         return bcd_ascii[c];
57:     }
58:
59:     inline intToInt() {
60:         return c;
61:     }
62:
63: };
64:
65: #endif
```

```
1: //  
2: // Implementation of bcd characters  
3: //  
4:  
5: #include "ubcd.h"  
6:  
7: /*  
8:     The following tables were copied from Joseph  
9:     Newcomer's 1401 emulation, in order to provide  
10:    consistent card, tape and print facilities.  
11:  
12:   Jay Jaeger, 12/96  
13: */  
14:  
15: /* The following table is given in the order of the 1401 BCD codes, and  
16:    contains the equivalent ASCII codes for printout.  
17: */  
18: int bcd_ascii[64] = {  
19:     ' ', /* 0           - space */  
20:     '1', /* 1           1   - 1 */  
21:     '2', /* 2           2   - 2 */  
22:     '3', /* 3           21  - 3 */  
23:     '4', /* 4           4   - 4 */  
24:     '5', /* 5           4 1  - 5 */  
25:     '6', /* 6           42  - 6 */  
26:     '7', /* 7           421 - 7 */  
27:     '8', /* 8           8   - 8 */  
28:     '9', /* 9           8 1  - 9 */  
29:     '0', /* 10          8 2  - 0 */  
30:     '=' , /* 11          8 21 - number sign (#) or equal */  
31:     '\'', /* 12          84   - at sign @ or quote */  
32:     ':', /* 13          84 1 - colon */  
33:     '>', /* 14          842  - greater than */  
34:     'û', /* 15          8421 - radical */  
35:     'b', /* 16          A    - substitute blank */  
36:     '/', /* 17          A 1  - slash */  
37:     'S', /* 18          A 2  - S */  
38:     'T', /* 19          A 21 - T */  
39:     'U', /* 20          A 4  - U */  
40:     'V', /* 21          A 4 1 - V */  
41:     'W', /* 22          A 42 - W */  
42:     'X', /* 23          A 421 - X */  
43:     'Y', /* 24          A8   - Y */  
44:     'Z', /* 25          A8 1 - Z */  
45:     'Ø', /* 26          A8 2 - record mark */  
46:     ',', /* 27          A8 21 - comma */  
47:     '(', /* 28          A84  - percent % or paren */  
48:     '^', /* 29          A84 1 - word separator */  
49:     '\\\\', /* 30          A842 - left oblique */  
50:     'x', /* 31          A8421 - segment mark */  
51:     '-' , /* 32          B    - hyphen */  
52:     'J', /* 33          B 1  - J */  
53:     'K', /* 34          B 2  - K */  
54:     'L', /* 35          B 21 - L */  
55:     'M', /* 36          B 4  - M */  
56:     'N', /* 37          B 4 1 - N */  
57:     'O', /* 38          B 42 - O */  
58:     'P', /* 39          B 421 - P */  
59:     'Q', /* 40          B 8  - Q */  
60:     'R', /* 41          B 8 1 - R */  
61:     '!', /* 42          B 8 2 - exclamation (-0) */  
62:     '$', /* 43          B 8 21 - dollar sign */  
63:     '*', /* 44          B 84  - asterisk */  
64:     ']', /* 45          B 84 1 - right bracket */  
65:     ';', /* 46          B 842 - semicolon */  
66:     '\177', /* 47          B 8421 - delta */
```

```

67:     '+' ,    /* 48 BA      - ampersand or plus */
68:     'A' ,    /* 49 BA 1   - A */
69:     'B' ,    /* 50 BA 2   - B */
70:     'C' ,    /* 51 BA 21  - C */
71:     'D' ,    /* 52 BA 4   - D */
72:     'E' ,    /* 53 BA 4 1 - E */
73:     'F' ,    /* 54 BA 42  - F */
74:     'G' ,    /* 55 BA 421 - G */
75:     'H' ,    /* 56 BA8   - H */
76:     'I' ,    /* 57 BA8 1  - I */
77:     '?' ,    /* 58 BA8 2  - question mark */
78:     '.' ,    /* 59 BA8 21 - period */
79:     ')' ,    /* 60 BA84  - lozenge or paren */
80:     '[' ,    /* 61 BA84 1 - left bracket */
81:     '<' ,   /* 62 BA842 - less than */
82:     ']' ,    /* 63 BA8421 - group mark */

83: };
84:
85:
86: ****
87: The following table is used to convert ASCII characters to BCD.
88:
89: Note that it currently is not complete.
90:
91: The following substitutions or alternate mappings are made:
92:
93:     ASCII code   BCD      Notes
94:     -----   ---      -----
95:     "        N/A      illegal
96:     %        (        'H' character set representation
97:     &       +        'H' character set representation
98:     @        '        'H' character set representation
99:     #        =        'H' character set representation
100:    ~       N/A      illegal
101:    ^       ^        substitute graphic for word-separator
102:    `       N/A      illegal
103:    a,c-z   A,C-Z    case folded
104:    b       b        substitute blank
105:    {       N/A      illegal
106:    }       N/A      illegal
107:    ~       N/A      illegal
108:    |       Ø        substitute for record mark
109:
110: ****
111:
112: int ascii_bcd[256] = {
113:     -1,-1,-1,-1,-1,-1,-1,    /* 00 - 07 illegal */
114:     -1,-1,-1,-1,-1,-1,-1,    /* 010 - 017 illegal */
115:     -1,-1,-1,-1,-1,-1,-1,    /* 020 - 027 illegal */
116:     -1,-1,-1,-1,-1,-1,-1,    /* 030 - 037 illegal */
117:
118:     0,          /* 040 space */
119:     42,         /* 041 ! */
120:     -1,         /* 042 " illegal */
121:     11,         /* 043 # */
122:     43,         /* 044 $ */
123:     28,         /* 045 % also ( */
124:     48,         /* 046 & also + */
125:     12,         /* 047 ' also @ */
126:
127:     28,         /* 050 ( also % */
128:     60,         /* 051 ) also lozenge */
129:     44,         /* 052 * */
130:     48,         /* 053 + also & */
131:     27,         /* 055 , */
132:     32,         /* 055 - */

```

```
133:      59,          /* 056 . */
134:      17,          /* 057 / */
135:
136:      10,          /* 060 0 */
137:      1,           /* 061 1 */
138:      2,           /* 062 2 */
139:      3,           /* 063 3 */
140:      4,           /* 064 4 */
141:      5,           /* 065 5 */
142:      6,           /* 066 6 */
143:      7,           /* 067 7 */
144:
145:      8,           /* 070 8 */
146:      9,           /* 071 9 */
147:     13,           /* 072 : */
148:     46,           /* 073 ; */
149:     62,           /* 074 < */
150:     11,           /* 075 = also # */
151:     14,           /* 076 > */
152:     58,           /* 077 ? */
153:
154:     12,           /* 0100 @ */
155:     49,           /* 0101 A */
156:     50,           /* 0102 B */
157:     51,           /* 0103 C */
158:     52,           /* 0104 D */
159:     53,           /* 0105 E */
160:     54,           /* 0106 F */
161:     55,           /* 0107 G */
162:
163:     56,           /* 0110 H */
164:     57,           /* 0111 I */
165:     33,           /* 0112 J */
166:     34,           /* 0113 K */
167:     35,           /* 0114 L */
168:     36,           /* 0115 M */
169:     37,           /* 0116 N */
170:     38,           /* 0117 O */
171:
172:     39,           /* 0120 P */
173:     40,           /* 0121 Q */
174:     41,           /* 0122 R */
175:     18,           /* 0123 S */
176:     19,           /* 0124 T */
177:     20,           /* 0125 U */
178:     21,           /* 0126 V */
179:     22,           /* 0127 W */
180:
181:     23,           /* 0130 X */
182:     24,           /* 0131 Y */
183:     25,           /* 0132 Z */
184:     61,           /* 0133 [ */
185:     30,           /* 0134 \ */
186:     45,           /* 0135 ] */
187:     29,           /* 0136 ^ word separator */
188:    -1,            /* 0137 _ illegal */
189:
190:    -1,            /* 0140 ` illegal */
191:    49,            /* 0141 a is A */
192:    16,            /* 0142 b is substitute blank */
193:    51,            /* 0143 c is C */
194:    52,            /* 0144 d is D */
195:    53,            /* 0145 e is E */
196:    54,            /* 0146 f is F */
197:    55,            /* 0147 g is G */
198:
```

```
199:      56,          /* 0150 h is H */
200:      57,          /* 0151 i is I */
201:      33,          /* 0152 j is J */
202:      34,          /* 0153 k is K */
203:      35,          /* 0154 l is L */
204:      36,          /* 0155 m is M */
205:      37,          /* 0156 n is N */
206:      38,          /* 0157 o is O */
207:
208:      39,          /* 0160 p is P */
209:      40,          /* 0161 q is Q */
210:      41,          /* 0162 r is R */
211:      18,          /* 0163 s is S */
212:      19,          /* 0164 t is T */
213:      20,          /* 0165 u is U */
214:      21,          /* 0166 v is V */
215:      22,          /* 0167 w is W */
216:
217:      23,          /* 0170 x is X */
218:      24,          /* 0171 y is Y */
219:      25,          /* 0172 z is Z */
220:      -1,          /* 0173 { illegal */
221:      26,          /* 0174 | substitute record mark */
222:      -1,          /* 0175 } illegal */
223:      -1,          /* 0176 ~ illegal */
224:      47,          /* 0177 □ delta */
225:
226:      -1,-1,-1,-1,-1,-1,-1,-1,    /* 0200-0207 illegal */
227:      -1,-1,-1,-1,-1,-1,-1,-1,    /* 0210-0217 illegal */
228:      -1,-1,-1,-1,-1,-1,-1,-1,    /* 0220-0227 illegal */
229:      -1,-1,-1,-1,-1,-1,-1,-1,    /* 0230-0237 illegal */
230:      -1,-1,-1,-1,-1,-1,-1,-1,    /* 0240-0247 illegal */
231:      -1,-1,-1,-1,-1,-1,-1,-1,    /* 0250-0257 illegal */
232:      -1,-1,-1,-1,-1,-1,-1,-1,    /* 0260-0267 illegal */
233:      -1,-1,-1,-1,-1,-1,-1,-1,    /* 0270-0277 illegal */
234:      -1,-1,-1,-1,-1,-1,-1,-1,    /* 0300-0307 illegal */
235:
236:      -1,-1,-1,-1,-1,-1,          /* 0310-0315 illegal */
237:      63,           /* 0316 group mark */
238:      -1,           /* 0317 illegal */
239:
240:      -1,-1,-1,-1,-1,-1,-1,      /* 0320-0326 illegal */
241:      31,           /* 0327 segment mark */
242:
243:      26,           /* 0330 record mark */
244:      -1,-1,-1,-1,-1,-1,-1,      /* 0331-0337 illegal */
245:
246:      -1,-1,-1,-1,-1,-1,-1,-1,    /* 0340-0347 illegal */
247:      -1,-1,-1,-1,-1,-1,-1,-1,    /* 0350-0357 illegal */
248:      -1,-1,-1,-1,-1,-1,-1,-1,    /* 0360-0367 illegal */
249:      -1,-1,-1,           /* 0370-0372 illegal */
250:      15,           /* 373 radical */
251:      -1,-1,-1,-1};           /* 0374-0377 illegal */
252:
253:
```

```
1: //  
2: // This Unit is the main part of the 1410 emulator  
3: //  
4:  
5: //-----  
6: #include <vc1\vc1.h>  
7: #pragma hdrstop  
8:  
9: #include "UI1410CPU.h"  
10: #include "UI1415IO.h"  
11: //-----  
12:  
13: // We have to predefine CPU, because the CPU object's children need a  
14: // pointer to the CPU to set up the various lists.  
15:  
16: T1410CPU *CPU = 0;  
17:  
18: // Construct and initialize the CPU.  
19:  
20: void Init1410()  
21: {  
22:     DEBUG("Creating CPU Object",0);  
23:  
24:     new T1410CPU;  
25:  
26:     // The following sets are for testing the Computer Reset button...  
27:  
28:     CPU -> IRing -> Set(I_RING_2);  
29:     CPU -> SubScanRing -> Set(SUB_SCAN_MQ);  
30:     CPU -> CycleRing -> Set(CYCLE_F);  
31:     CPU -> CarryIn -> Set();  
32:     CPU -> BComplement -> Set();  
33:     CPU -> CompareBLTA -> Reset();  
34:     CPU -> CompareBEQA -> Set();  
35:  
36:     // End of test sets  
37:  
38:     FI1415IO -> SetState(CONSOLE_IDLE);  
39:     CPU -> Display();  
40:  
41:     DEBUG("Waiting for User Interface",0)  
42:  
43: }  
44:
```

```
1: //-----
2: #ifndef UI1410CPUH
3: #define UI1410CPUH
4:
5: #include "ubcd.h"
6:
7: //
8: // Classes (types) used to implement the emulator, including the
9: // final class defining what is in the CPU, T1410CPU.
10:
11: //
12: // Abstract class designed to build lists of objects affected by Program
13: // Reset and Computer Reset
14: //
15:
16: class TCpuObject : public TObject {
17: public:
18:     TCpuObject();                                // Constructor to init data
19:     virtual void OnComputerReset() = 0;           // Called during Computer Reset
20:     virtual void OnProgramReset() = 0;             // Called during Program Reset
21:     void AddToComputerReset();                    // Called by constructor to add to list
22:     void AddToProgramReset();                     // Called by constructor to add to list
23:
24: public:
25:     TCpuObject *NextComputerReset;
26:     TCpuObject *NextProgramReset;
27: };
28:
29: //
30: // A second abstract class of objects that not only react to the Resets,
31: // but also have entries on the display panel.
32: //
33:
34: class TDisplayObject : public TCpuObject {
35: public:
36:     TDisplayObject();                            // Constructor to init data.
37:     virtual void Display() = 0;                  // Called to display this item
38:     virtual void LampTest(bool b) = 0;            // Called to start/end lamp test
39:
40: public:
41:     TDisplayObject *NextDisplay;
42: };
43:
44: class TRingCounter : public TDisplayObject {
45: private:
46:     char state;                                // State of ring counter (FF Number)
47:     char max;                                  // Max number of entries
48:     TLabel *lastlamp;                          // Last lamp to be displayed
49:
50: public:
51:     TLabel **lamps;                           // Ptr to array of lamps
52:
53: public:
54:     TRingCounter(char n);                     // Construct with # of entries
55:     virtual __fastcall ~TRingCounter();        // Destructor for array of lamps
56:
57: // Functions inherited from abstract base classes now need definition
58:
59: void OnComputerReset();
60: void OnProgramReset();
61: void Display();
62: void LampTest(bool b);
63:
64: // The real meat of the Ring Counter class
65:
66: inline void Reset() { state = 0; }
```

```
67:     inline char Set(char n) {return state = n; }
68:     inline char State() { return state; }
69:     char Next();
70: };
71:
72: // This next class is like the Ring Counters, except that it is not
73: // affected by Program Reset.
74:
75: class TDisplayLatch : public TDisplayObject {
76: private:
77:     bool state;                                // Latches can be set or reset
78:     TLabel *lamp;                             // Pointer to display lamp
79: public:
80:     TDisplayLatch(TLabel *l);                  // Constructor - set up lamp
81:
82:     // Functions inherited from abstract base class classes now need definition
83:
84:     void OnComputerReset();
85:     void AddToProgramReset();                  // Program Reset does not reset these!
86:     void OnProgramReset();                    // Program Reset does not reset these.
87:     void Display();
88:     void LampTest(bool b);
89:
90:     // All you can really do with latches is set/reset/test
91:
92:     inline void Reset() { state = false; }
93:     inline void Set()   { state = true; }
94:     inline void Set(bool b) { state = b; }
95:     inline bool State() { return state; }
96: };
97:
98: // This class defines what is in an I/O Channel
99:
100: #define IOCHNOTREADY    1
101: #define IOCHBUSY        2
102: #define IOCHDATACHECK  4
103: #define IOCHCONDITION  8
104: #define IOCHNOTTRANSFER 16
105: #define IOCHWLRECORD   32
106:
107: #define IOLAMPNOTREADY  0
108: #define IOLAMPBUSY      1
109: #define IOLAMPDATACHECK 2
110: #define IOLAMPCONDITION 3
111: #define IOLAMPNOTTRANSFER 4
112: #define IOLAMPWLRECORD  5
113:
114: class T1410Channel : public TDisplayObject {
115:
116: public:
117:
118:     // Functions inherited from abstract base class classes now need definition
119:
120:     void OnComputerReset();
121:     void AddToProgramReset();                // Program Reset does not reset these!
122:     void OnProgramReset();                  // Program Reset does not reset these.
123:     void Display();
124:     void LampTest(bool b);
125:
126: private:
127:
128:     // Channel information
129:
130:     int ChStatus;                          // Channel status (see defines)
131:
132:     TLabel *ChStatusDisplay[6];           // Channel status lights
```

```
133:  
134: public:  
135:  
136:     TDisplayLatch *ChInterlock;  
137:     TDisplayLatch *ChRBCInterlock;  
138:     TDisplayLatch *ChRead;  
139:     TDisplayLatch *ChWrite;  
140:     TDisplayLatch *ChOverlap;  
141:     TDisplayLatch *ChNotOverlap;  
142:  
143:     // Methods  
144:  
145:     T1410Channel(                                // Constructor  
146:         TLabel *LampInterlock,  
147:         TLabel *LampRBCInterlock,  
148:         TLabel *LampRead,  
149:         TLabel *LampWWrite,  
150:         TLabel *LampOverlap,  
151:         TLabel *LampNotOverlap,  
152:         TLabel *LampNotRead,  
153:         TLabel *LampBusy,  
154:         TLabel *LampDataCheck,  
155:         TLabel *LampCondition,  
156:         TLabel *LampWLRecord,  
157:         TLabel *LampNoTransfer  
158:     );  
159:  
160:     inline int SetStatus(int i) { return ChStatus = i; }  
161:     inline int GetStatus() { return ChStatus; }  
162: };  
163:  
164: // This class defines what is actually inside the CPU.  
165:  
166: #define MAXCHANNEL 2  
167: #define CHANNEL1 0  
168: #define CHANNEL2 1  
169:  
170: #define I_RING_OP 0  
171: #define I_RING_1 1  
172: #define I_RING_2 2  
173: #define I_RING_3 3  
174: #define I_RING_4 4  
175: #define I_RING_5 5  
176: #define I_RING_6 6  
177: #define I_RING_7 7  
178: #define I_RING_8 8  
179: #define I_RING_9 9  
180: #define I_RING_10 10  
181: #define I_RING_11 11  
182: #define I_RING_12 12  
183:  
184: #define A_RING_1 0  
185: #define A_RING_2 1  
186: #define A_RING_3 2  
187: #define A_RING_4 3  
188: #define A_RING_5 4  
189: #define A_RING_6 5  
190:  
191: #define CLOCK_A 0  
192: #define CLOCK_B 1  
193: #define CLOCK_C 2  
194: #define CLOCK_D 3  
195: #define CLOCK_E 4  
196: #define CLOCK_F 5  
197: #define CLOCK_G 6  
198: #define CLOCK_H 7
```

```
199: #define CLOCK_J 8
200: #define CLOCK_K 9
201:
202: #define SCAN_N 0
203: #define SCAN_1 1
204: #define SCAN_2 2
205: #define SCAN_3 3
206:
207: #define SUB_SCAN_NONE 0
208: #define SUB_SCAN_U 1
209: #define SUB_SCAN_B 2
210: #define SUB_SCAN_E 3
211: #define SUB_SCAN_MQ 4
212:
213: #define CYCLE_A 0
214: #define CYCLE_B 1
215: #define CYCLE_C 2
216: #define CYCLE_D 3
217: #define CYCLE_E 4
218: #define CYCLE_F 5
219: #define CYCLE_I 6
220: #define CYCLE_X 7
221:
222: class T1410CPU {
223:
224: public:
225:
226:     // Wiring list
227:
228:     TCpuObject *ComputerResetList;           // List of things affected
229:     TCpuObject *ProgramResetList;           // List of things affected
230:     TDisplayObject *DisplayList;            // List of displayable things
231:
232:     // Channels
233:
234:     T1410Channel *Channel[MAXCHANNEL];      // 2 I/O Channels.
235:
236:     // Registers and ring counters
237:
238:     TRingCounter *IRing;                   // Instruction decode ring
239:     TRingCounter *ARing;                  // Address decode ring
240:     TRingCounter *ClockRing;              // Cycle Clock
241:     TRingCounter *ScanRing;               // Address Modification Mode
242:     TRingCounter *SubScanRing;            // Arithmetic Scan type
243:     TRingCounter *CycleRing;              // CPU Cycle type
244:
245:     // Latches
246:
247:     TDisplayLatch *CarryIn;                // Carry latch
248:     TDisplayLatch *CarryOut;               // Adder has generated carry
249:     TDisplayLatch *AComplement;            // A channel complement
250:     TDisplayLatch *BComplement;            // B channel complement
251:     TDisplayLatch *CompareBGTA;           // B > A
252:     TDisplayLatch *CompareBEQA;           // B = A
253:     TDisplayLatch *CompareBLTA;           // B < A NOTE: On after C. Reset.
254:     TDisplayLatch *Overflow;              // Arithmetic Overflow
255:     TDisplayLatch *DivideOverflow;        // Divide Overflow
256:     TDisplayLatch *ZeroBalance;           // Zero arithmetic result
257:
258:     // Switches
259:
260:     enum Mode {                           // Mode switch, values must match
261:         MODE_RUN = 0, MODE_DISPLAY = 1, MODE_ALTER = 2,
262:         MODE_CE = 3, MODE_IE = 4, MODE_ADDR = 5
263:     } Mode;
264:
```

```
265:     enum AddressEntry {
266:         ADDR_ENTRY_I = 0, ADDR_ENTRY_A = 1, ADDR_ENTRY_B = 2,
267:         ADDR_ENTRY_C = 3, ADDR_ENTRY_D = 4, ADDR_ENTRY_E = 5,
268:         ADDR_ENTRY_F = 6
269:     } AddressEntry;
270:
271:     enum StorageScan {
272:         SSCAN_OFF = 0, SSCAN_LOAD_1 = 1, SSCAN_LOAD_0 = 2,
273:         SSCAN_REGEN_0 = 3, SSCAN_REGEN_1 = 4
274:     } StorageScan;
275:
276:     enum CycleControl {
277:         CYCLE_OFF = 0, CYCLE_LOGIC = 1, CYCLE_STORAGE = 2
278:     } CycleControl;
279:
280:     enum CheckControl {
281:         CHECK_STOP = 0, CHECK_RESTART = 1, CHECK_RESET = 2
282:     } CheckControl;
283:
284:     bool DiskWrInhibit;
285:
286:     enum TapeDensity {
287:         DENSITY_200_556 = 0, DENSITY_200_800 = 1, DENSITY_556_800 = 2
288:     } DensityCh1, DensityCh2; Tape Density
289:
290:     bool AsteriskInsert;
291:     bool InhibitPrintOut;
292:
293:     BCD BitSwitches;
294:
295: // Methods
296:
297:     T1410CPU(); // Constructor
298:     void Display(); // Run thru the display list
299:
300: };
301:
302: extern T1410CPU *CPU;
303:
304: //-----
305: #endif
306:
```

*More 16 Channel*

```
1: // This Unit provides the functionality behind some of my private
2: // types for the 1410 emulator
3:
4: //-----
5: #include <vcl\vcl.h>
6: #include <assert.h>
7: #pragma hdrstop
8:
9: #include "UI1410CPUUT.h"
10: #include "UI1415L.h"
11:
12: //-----
13:
14: // Declarations for Borland VCL controls
15:
16: #include <vcl\Classes.hpp>
17: #include <vcl\Controls.hpp>
18: #include <vcl\StdCtrls.hpp>
19:
20: // Implementation of TCpuObject (Abstract Base Class)
21:
22: TCpuObject::TCpuObject()
23: {
24:     NextComputerReset = CPU -> ComputerResetList;
25:     CPU -> ComputerResetList = this;
26:     NextProgramReset = CPU -> ProgramResetList;
27:     CPU -> ProgramResetList = this;
28: }
29:
30: void TCpuObject::AddToComputerReset()
31: {
32:     NextComputerReset = CPU -> ComputerResetList;
33:     CPU -> ComputerResetList = this;
34: }
35:
36: void TCpuObject::AddToProgramReset()
37: {
38:     NextProgramReset = CPU -> ProgramResetList;
39:     CPU -> ProgramResetList = this;
40: }
41:
42: // Implementation of TDisplayObject (Abstract Base Class)
43:
44: TDisplayObject::TDisplayObject()
45: {
46:     NextDisplay = CPU -> DisplayList;
47:     CPU -> DisplayList = this;
48: }
49:
50: // Implementation of TRingCounter (Ring Counter Class)
51:
52: // The constructor sets the max state of the ring, and resets the ring
53: // It also allocates an array of pointers to the lamps. The creator must
54: // fill in that array, however. Initially, the lamp pointers are empty,
55: // (which means no lamp is attached to that state).
56:
57: TRingCounter::TRingCounter(char n)
58: {
59:     int i;
60:
61:     state = 0;
62:     max = n;
63:     lastlamp = 0;
64:
65:     lamps = new TLabel*[n];
66:     for(i=0; i < n; ++i) {
```

```
67:         lamps[i] = 0;
68:     }
69: }
70:
71: // The destructor frees up the array of lamps. Probably will never use.
72:
73: __fastcall TRingCounter::~TRingCounter()
74: {
75:     delete[] lamps;
76: }
77:
78: // All Ring counters are reset on PROGRAM and COMPUTER RESET
79:
80: void TRingCounter::OnComputerReset()
81: {
82:     state = 0;
83: }
84:
85: void TRingCounter::OnProgramReset()
86: {
87:     state = 0;
88: }
89:
90: // When the Display routine is called, it resets the lamp corresponding
91: // to the state when it last displayed, and then displays the current state
92: // If there is no lamp associated with a state (lamp pointer is null),
93: // then don't display any lamp.
94:
95: void TRingCounter::Display()
96: {
97:     if(lastlamp) {
98:         lastlamp -> Enabled = false;
99:         lastlamp -> Repaint();
100:    }
101:    if(lamps[state] == 0) {
102:        lastlamp = 0;
103:        return;
104:    }
105:    lamps[state] -> Enabled = true;
106:    lamps[state] -> Repaint();
107:    lastlamp = lamps[state];
108: }
109:
110: // On a lamp test, light all the associated lamps.
111: // On reset of a lamp test, clear them all, then display the current state.
112:
113: void TRingCounter::LampTest(bool b)
114: {
115:     int i;
116:
117:     for(i = 0; i < max; ++i) {
118:         if(lamps[i] != 0) {
119:             lamps[i] -> Enabled = b;
120:             lamps[i] -> Repaint();
121:         }
122:     }
123:     if(!b) {
124:         lastlamp = 0;
125:         this -> Display();
126:     }
127: }
128:
129: // Next advances to the next state (or back to the start if appropriate)
130: // Returns current state. This is only useful for true Ring counters.
131:
132: char TRingCounter::Next()
```

```
133: {
134:     if(++state >= max) {
135:         state = 0;
136:     }
137:     return(state);
138: }
139:
140: // Implementation of TDisplayLatch (Display Latch Class)
141:
142: // The constructor initializes the latch and sets the pointer to a lamp.
143: // A pointer to a lamp is required.
144:
145: TDisplayLatch::TDisplayLatch(TLabel *l)
146: {
147:     state = false;
148:     lamp = l;
149: }
150:
151: // All Displayable Latches are reset on a COMPUTER RESET
152:
153: void TDisplayLatch::OnComputerReset()
154: {
155:     state = false;
156: }
157:
158: // Displayable Latches are *NOT* reset on PROGRAM RESET, so don't add to list
159:
160: void TDisplayLatch::AddToProgramReset()
161: {
162:     // Display Latches are *not* reset, so don't add them to the list.
163: }
164:
165: void TDisplayLatch::OnProgramReset()
166: {
167:     // Display Latches are *not* reset by a program reset.
168: }
169:
170: // When the Display routine is called, it sets or resets the lamp,
171: // depending on the current state.
172:
173: void TDisplayLatch::Display()
174: {
175:     lamp -> Enabled = state;
176:     lamp -> Repaint();
177: }
178:
179: // On a lamp test, light all the lamp.
180: // On reset of a lamp test, display the current state.
181:
182: void TDisplayLatch::LampTest(bool b)
183: {
184:     lamp -> Enabled = (b ? true : state);
185:     lamp -> Repaint();
186: }
187:
188: // Implementation of I/O Channel Class
189:
190: // Constructor.  Initializes state
191:
192: T1410Channel::T1410Channel(
193:     TLabel *LampInterlock,
194:     TLabel *LampRBCInterlock,
195:     TLabel *LampRead,
196:     TLabel *LampWrite,
197:     TLabel *LampOverlap,
198:     TLabel *LampNotOverlap,
```

```
199:         TLabel *LampNotReady,
200:         TLabel *LampBusy,
201:         TLabel *LampDataCheck,
202:         TLabel *LampCondition,
203:         TLabel *LampWLRecord,
204:         TLabel *LampNoTransfer ) {
205:
206:     ChStatus = 0;
207:
208:     ChStatusDisplay[IOLAMPNOTREADY] = LampNotReady;
209:     ChStatusDisplay[IOLAMPBUSY] = LampBusy;
210:     ChStatusDisplay[IOLAMPDATACHECK] = LampDataCheck;
211:     ChStatusDisplay[IOLAMPCONDITION] = LampCondition;
212:     ChStatusDisplay[IOLAMPNOTTRANSFER] = LampNoTransfer;
213:     ChStatusDisplay[IOLAMPWLRECORD] = LampWLRecord;
214:
215:     ChInterlock = new TDisplayLatch(LampInterlock);
216:     ChRBCInterlock = new TDisplayLatch(LampRBCInterlock);
217:     ChRead = new TDisplayLatch(LampRead);
218:     ChWrite = new TDisplayLatch(LampWrite);
219:     ChOverlap = new TDisplayLatch(LampOverlap);
220:     ChNotOverlap = new TDisplayLatch(LampNotOverlap);
221: }
222:
223: // Channel is reset during ComputerReset
224:
225: void T1410Channel::OnComputerReset()
226: {
227:     ChStatus = 0;
228:
229:     // Note: The objects which are TDisplayLatch objects will reset themselves
230: }
231:
232: // Channel Latches are *NOT* reset on PROGRAM RESET, so don't add to list
233:
234: void T1410Channel::AddToProgramReset()
235: {
236:     // Channel Latches are *not* reset, so don't add them to the list.
237: }
238:
239: void T1410Channel::OnProgramReset()
240: {
241:     // Channel Latches are *not* reset by a program reset.
242: }
243:
244: // Display Routine.
245:
246: void T1410Channel::Display() {
247:
248:     int i;
249:
250:     ChStatusDisplay[IOLAMPNOTREADY] -> Enabled =
251:         ((ChStatus & IOCHNOTREADY) != 0);
252:     ChStatusDisplay[IOLAMPBUSY] -> Enabled =
253:         ((ChStatus & IOCHBUSY) != 0);
254:     ChStatusDisplay[IOLAMPDATACHECK] -> Enabled =
255:         ((ChStatus & IOCHDATACHECK) != 0);
256:     ChStatusDisplay[IOLAMPCONDITION] -> Enabled =
257:         ((ChStatus & IOCHCONDITION) != 0);
258:     ChStatusDisplay[IOLAMPWLRECORD] -> Enabled =
259:         ((ChStatus & IOCHWLRECORD) != 0);
260:     ChStatusDisplay[IOLAMPNOTTRANSFER] -> Enabled =
261:         ((ChStatus & IOCHNOTTRANSFER) != 0);
262:
263:     for(i=0; i <= 5; ++i) {
264:         ChStatusDisplay[i] -> Repaint();
```

```
265:     }
266:
267: // Although in most instances the following would be redundant,
268: // because these objects are also on the CPU display list, we include
269: // them here in case we want to display a channel separately.
270:
271: ChInterlock -> Display();
272: ChRBCInterlock -> Display();
273: ChRead -> Display();
274: ChWrite -> Display();
275: ChOverlap -> Display();
276: ChNotOverlap -> Display();
277: }
278:
279: // Channel Lamp Test
280:
281: void T1410Channel::LampTest(bool b)
282: {
283:     int i;
284:
285: // Note, we don't have to do anything to the TDisplayLatch objects in
286: // the channel for lamp test. They will take care of themselves on a
287: // lamp test.
288:
289: if(!b) {
290:     for(i=0; i <= 5; ++i) {
291:         ChStatusDisplay[i] -> Enabled = true;
292:         ChStatusDisplay[i] -> Repaint();
293:     }
294: } else {
295:     Display();
296: }
297: }
298: }
299:
300: // CPU object constructor. Essentially this method "wires" the 1410.
301:
302: T1410CPU::T1410CPU()
303: {
304:     CPU = this;
305:
306: // Clear out the lists
307:
308:     DisplayList = 0;
309:     ComputerResetList = 0;
310:     ProgramResetList = 0;
311:
312: // Set switches to initial states
313:
314:     Mode = MODE_RUN;
315:     AddressEntry = ADDR_ENTRY_I;
316:     StorageScan = SSCAN_OFF;
317:     CycleControl = CYCLE_OFF;
318:     CheckControl = CHECK_STOP;
319:     DiskWrInhibit = false;
320:     DensityCh1 = DensityCh2 = DENSITY_200_556; )more to Channel
321:     AsteriskInsert = true;
322:     InhibitPrintOut = false;
323:     BitSwitches = BCD(0);
324:
325: // Build the various displayable components of the CPU
326:
327:     IRing = new TRingCounter(13);
328:     assert(F1415L -> Light_I_OP != 0);
329:     IRing -> lamps[0] = F1415L -> Light_I_OP;
330:     IRing -> lamps[1] = F1415L -> Light_I_1;
```

```
331:     IRing -> lamps[2] = F1415L -> Light_I_2;
332:     IRing -> lamps[3] = F1415L -> Light_I_3;
333:     IRing -> lamps[4] = F1415L -> Light_I_4;
334:     IRing -> lamps[5] = F1415L -> Light_I_5;
335:     IRing -> lamps[6] = F1415L -> Light_I_6;
336:     IRing -> lamps[7] = F1415L -> Light_I_7;
337:     IRing -> lamps[8] = F1415L -> Light_I_8;
338:     IRing -> lamps[9] = F1415L -> Light_I_9;
339:     IRing -> lamps[10] = F1415L -> Light_I_10;
340:     IRing -> lamps[11] = F1415L -> Light_I_11;
341:     IRing -> lamps[12] = F1415L -> Light_I_12;
342:
343:     ARing = new TRingCounter(6);
344:     ARing -> lamps[0] = F1415L -> Light_A_1;
345:     ARing -> lamps[1] = F1415L -> Light_A_2;
346:     ARing -> lamps[2] = F1415L -> Light_A_3;
347:     ARing -> lamps[3] = F1415L -> Light_A_4;
348:     ARing -> lamps[4] = F1415L -> Light_A_5;
349:     ARing -> lamps[5] = F1415L -> Light_A_6;
350:
351:     ClockRing = new TRingCounter(10);
352:     ClockRing -> lamps[0] = F1415L -> Light_Clk_A;
353:     ClockRing -> lamps[1] = F1415L -> Light_Clk_B;
354:     ClockRing -> lamps[2] = F1415L -> Light_Clk_C;
355:     ClockRing -> lamps[3] = F1415L -> Light_Clk_D;
356:     ClockRing -> lamps[4] = F1415L -> Light_Clk_E;
357:     ClockRing -> lamps[5] = F1415L -> Light_Clk_F;
358:     ClockRing -> lamps[6] = F1415L -> Light_Clk_G;
359:     ClockRing -> lamps[7] = F1415L -> Light_Clk_H;
360:     ClockRing -> lamps[8] = F1415L -> Light_Clk_J;
361:     ClockRing -> lamps[9] = F1415L -> Light_Clk_K;
362:
363:     ScanRing = new TRingCounter(4);
364:     ScanRing -> lamps[0] = F1415L -> Light_Scan_N;
365:     ScanRing -> lamps[1] = F1415L -> Light_Scan_1;
366:     ScanRing -> lamps[2] = F1415L -> Light_Scan_2;
367:     ScanRing -> lamps[3] = F1415L -> Light_Scan_3;
368:
369:     SubScanRing = new TRingCounter(5);
370:     // NOTE: State 0 is "OFF" - no flip flops set
371:     SubScanRing -> lamps[1] = F1415L -> Light_Sub_Scan_U;
372:     SubScanRing -> lamps[2] = F1415L -> Light_Sub_Scan_B;
373:     SubScanRing -> lamps[3] = F1415L -> Light_Sub_Scan_E;
374:     SubScanRing -> lamps[4] = F1415L -> Light_Sub_Scan_MQ;
375:
376:     CycleRing = new TRingCounter(8);
377:     CycleRing -> lamps[0] = F1415L -> Light_Cycle_A;
378:     CycleRing -> lamps[1] = F1415L -> Light_Cycle_B;
379:     CycleRing -> lamps[2] = F1415L -> Light_Cycle_C;
380:     CycleRing -> lamps[3] = F1415L -> Light_Cycle_D;
381:     CycleRing -> lamps[4] = F1415L -> Light_Cycle_E;
382:     CycleRing -> lamps[5] = F1415L -> Light_Cycle_F;
383:     CycleRing -> lamps[6] = F1415L -> Light_Cycle_I;
384:     CycleRing -> lamps[7] = F1415L -> Light_Cycle_X;
385:
386:     CarryIn = new TDisplayLatch(F1415L -> Light_Carry_In);
387:     CarryOut = new TDisplayLatch(F1415L -> Light_Carry_Out);
388:     AComplement = new TDisplayLatch(F1415L -> Light_A_Complement);
389:     BComplement = new TDisplayLatch(F1415L -> Light_B_Complement);
390:     CompareBGTA = new TDisplayLatch(F1415L -> Light_B_GT_A);
391:     CompareBEQA = new TDisplayLatch(F1415L -> Light_B_EQ_A);
392:     CompareBLTA = new TDisplayLatch(F1415L -> Light_B_LT_A);
393:     Overflow = new TDisplayLatch(F1415L -> Light_Overflow);
394:     DivideOverflow = new TDisplayLatch(F1415L -> Light_Divide_Overflow);
395:     ZeroBalance = new TDisplayLatch(F1415L -> Light_Zero_Balance);
396:
```

```
397:     // Build the channels
398:
399:     Channel[CHANNEL1] = new T1410Channel(
400:         F1415L -> Light_Ch1_Interlock,
401:         F1415L -> Light_Ch1_RBCInterlock,
402:         F1415L -> Light_Ch1_Read,
403:         F1415L -> Light_Ch1_Write,
404:         F1415L -> Light_Ch1_Overlap,
405:         F1415L -> Light_Ch1_NoOverlap,
406:         F1415L -> Light_Ch1_NotReady,
407:         F1415L -> Light_Ch1_Busy,
408:         F1415L -> Light_Ch1_DataCheck,
409:         F1415L -> Light_Ch1_Condition,
410:         F1415L -> Light_Ch1_WLRecord,
411:         F1415L -> Light_Ch1_NoTransfer
412:     );
413:
414:     Channel[CHANNEL2] = new T1410Channel(
415:         F1415L -> Light_Ch2_Interlock,
416:         F1415L -> Light_Ch2_RBCInterlock,
417:         F1415L -> Light_Ch2_Read,
418:         F1415L -> Light_Ch2_Write,
419:         F1415L -> Light_Ch2_Overlap,
420:         F1415L -> Light_Ch2_NoOverlap,
421:         F1415L -> Light_Ch2_NotReady,
422:         F1415L -> Light_Ch2_Busy,
423:         F1415L -> Light_Ch2_DataCheck,
424:         F1415L -> Light_Ch2_Condition,
425:         F1415L -> Light_Ch2_WLRecord,
426:         F1415L -> Light_Ch2_NoTransfer
427:     );
428:
429:     // Some latches are set after power on...
430:
431:     CompareBLTA -> Set();
432:
433: }
434:
435:
436: // CPU Object display - run thru the display list
437:
438: void T1410CPU::Display()
439: {
440:     TDisplayObject *l;
441:
442:     for(l = DisplayList; l != 0; l = l -> NextDisplay) {
443:         l -> Display();
444:     }
445: }
```

```
1: //-----
2: #ifndef UI1415CEH
3: #define UI1415CEH
4: //-----
5: #include <vcl\Classes.hpp>
6: #include <vcl\Controls.hpp>
7: #include <vcl\StdCtrls.hpp>
8: #include <vcl\Forms.hpp>
9: //-----
10: class TFI1415CE : public TForm
11: {
12:     __published:    // IDE-managed Components
13:         TComboBox *AddressEntry;
14:         TLabel *Label1;
15:         TComboBox *StorageScan;
16:         TLabel *Label2;
17:         TComboBox *CycleControl;
18:         TLabel *Label3;
19:         TComboBox *CheckControl;
20:         TLabel *Label4;
21:         TCheckBox *DiskWrInhibit;
22:         TComboBox *DensityCh1;
23:         TLabel *Label5;
24:         TComboBox *DensityCh2;
25:         TLabel *Label6;
26:         TButton *StartPrintOut;
27:         TCheckBox *Compat1401;
28:         TButton *CheckReset1401;
29:         TCheckBox *CheckStop1401;
30:         TButton *CheckTest1;
31:         TButton *CheckTest2;
32:         TButton *CheckTest3;
33:         TLabel *Label7;
34:         TLabel *Label8;
35:         TCheckBox *AsteriskInsert;
36:         TCheckBox *InhibitPrintOut;
37:         TCheckBox *BitSenseC;
38:         TCheckBox *BitSenseB;
39:         TCheckBox *BitSenseA;
40:         TCheckBox *BitSense8;
41:         TCheckBox *BitSense4;
42:         TCheckBox *BitSense2;
43:         TCheckBox *BitSense1;
44:         TCheckBox *BitSenseWM;
45:         TLabel *Label9;
46:         TLabel *Label10;
47:         TLabel *Label11;
48:         void __fastcall AddressEntryChange(TObject *Sender);
49:
50:         void __fastcall StorageScanChange(TObject *Sender);
51:         void __fastcall CycleControlChange(TObject *Sender);
52:         void __fastcall CheckControlChange(TObject *Sender);
53:         void __fastcall DiskWrInhibitClick(TObject *Sender);
54:         void __fastcall DensityCh1Change(TObject *Sender);
55:         void __fastcall DensityCh2Change(TObject *Sender);
56:         void __fastcall AsteriskInsertClick(TObject *Sender);
57:         void __fastcall InhibitPrintOutClick(TObject *Sender);
58:         void __fastcall BitSenseCClick(TObject *Sender);
59:         void __fastcall BitSenseBClick(TObject *Sender);
60:         void __fastcall BitSenseAClick(TObject *Sender);
61:         void __fastcall BitSense8Click(TObject *Sender);
62:         void __fastcall BitSense4Click(TObject *Sender);
63:         void __fastcall BitSense2Click(TObject *Sender);
64:         void __fastcall BitSense1Click(TObject *Sender);
65:         void __fastcall BitSenseWMClick(TObject *Sender);
66:     private:    // User declarations
```

```
67:     int SenseBit;
68:     int SetSense(bool b,int i);
69: public:      // User declarations
70:     __fastcall TFI1415CE(TComponent* Owner);
71: };
72: //-----
73: extern TFI1415CE *FI1415CE;
74: //-----
75: #endif
```

```
1: //-----
2: #ifndef UI1415IOH
3: #define UI1415IOH
4: //-----
5: #include <vcl\Classes.hpp>
6: #include <vcl\Controls.hpp>
7: #include <vcl\StdCtrls.hpp>
8: #include <vcl\Forms.hpp>
9: //-----
10:
11: // 1415 Console User Interface
12:
13: #include "ubcd.h"
14: #include <vcl\ExtCtrls.hpp>
15:
16: #define CONSOLE_IDLE      1           // Console is idle
17: #define CONSOLE_NORMAL    2           // Normal Read Mode input
18: #define CONSOLE_LOAD      3           // Normal Load Mode input
19: #define CONSOLE_ALTER     4           // Console is in alter mode loading memory
20: #define CONSOLE_ADDR      5           // Console is in address set or display mode
21:
22: class TFI1415IO : public TForm
23: {
24:     __published: // IDE-managed Components
25:         TMemo *I1415IO;
26:         TLabel *KeyboardLock;
27:         TButton *InqReq;
28:         TButton *InqRlse;
29:         TButton *InqCancel;
30:         TButton *WordMark;
31:         TButton *MarginRelease;
32:         TTimer *KeyboardLockReset;
33:         void __fastcall I1415IOKeyPress(TObject *Sender, char &Key);
34:
35:         void __fastcall I1415IOKeyDown(TObject *Sender, WORD &Key, TShiftState Shift);
36:
37:         void __fastcall WordMarkClick(TObject *Sender);
38:         void __fastcall MarginReleaseClick(TObject *Sender);
39:
40:         void __fastcall KeyboardLockResetTimer(TObject *Sender);
41: private: // User declarations
42:     int state;
43:     void __fastcall NextLine();
44:     void __fastcall DoWordMark();
45:     void __fastcall LockKeyboard();
46:     void __fastcall UnlockKeyboard();
47: public: // User declarations
48:     __fastcall TFI1415IO(TComponent* Owner);
49:     void __fastcall SendBCDTo1415(BCD bcd);
50:     bool __fastcall SetState(int s);
51: };
52: //-----
53: extern TFI1415IO *FI1415IO;
54: //-----
55:
56: // Keyboard representations of some unusual BCD characters
57:
58: #define KBD_RADICAL          022
59: #define KBD_RECORD_MARK       '|'
60: #define KBD_ALT_BLANK         002
61: #define KBD_WORD_SEPARATOR    '^'
62: #define KBD_SEGMENT_MARK     023
63: #define KBD_DELTA              004
64: #define KBD_GROUP_MARK        007
65: #define KBD_WORD_MARK         0x1b
66:
```

67: #endif

```
1: // This Unit defines the behavior of the 1415 console typewriter and
2: // associated things
3:
4: //-----
5: #include <vcl\vcl.h>
6: #pragma hdrstop
7:
8: #include "UI1415IO.h"
9: #include "ubcd.h"
10:
11: //-----
12: #pragma resource "* .dfm"
13: TFI1415IO *FI1415IO;
14: //-----
15: __fastcall TFI1415IO::TFI1415IO(TComponent* Owner)
16:   : TForm(Owner)
17: {
18:   state = CONSOLE_IDLE;
19: }
20: //-----
21: void __fastcall TFI1415IO::I1415IOKeyPress(TObject *Sender, char &Key)
22: {
23:   BCD bcd_key;
24:
25:   switch(Key) {
26:     case KBD_RADICAL:
27:       bcd_key = bcd_key.BCDConvert(ASCII_RADICAL);
28:       break;
29:     case KBD_RECORD_MARK:
30:       bcd_key = bcd_key.BCDConvert(ASCII_RECORD_MARK);
31:       break;
32:     case KBD_ALT_BLANK:
33:       bcd_key = bcd_key.BCDConvert(ASCII_ALT_BLANK);
34:       break;
35:     case KBD_WORD_SEPARATOR:
36:       bcd_key = bcd_key.BCDConvert(ASCII_WORD_SEPARATOR);
37:       break;
38:     case KBD_SEGMENT_MARK:
39:       bcd_key = bcd_key.BCDConvert(ASCII_SEGMENT_MARK);
40:       break;
41:     case KBD_DELTA:
42:       bcd_key = bcd_key.BCDConvert(ASCII_DELTA);
43:       break;
44:     case KBD_GROUP_MARK:
45:       bcd_key = bcd_key.BCDConvert(ASCII_GROUP_MARK);
46:       break;
47:     case KBD_WORD_MARK: // Escape == Wordmark Key
48:       DoWordMark();
49:       return;
50:     case 'b':
51:       bcd_key = bcd_key.BCDConvert('B');
52:       break;
53:     default:
54:       if(bcd_key.BCDCheck(Key) < 0) { // Return if unmapped key.
55:         LockKeyboard();
56:         return;
57:       }
58:       bcd_key = bcd_key.BCDConvert(Key);
59:   }
60:
61:   // Decide what to do with key, depending on console state.
62:   // Note that the really special keys (Wordmark, above, and the
63:   // console inquiry buttons), are checked elsewhere. This test
64:   // is for "normal" BCD characters only.
65:
66:   switch(state) {
```

```
67:     case CONSOLE_IDLE:
68:         LockKeyboard();                                // Only INQ Request allowed
69:         return;
70:     case CONSOLE_NORMAL:                           // All keys allowed
71:     case CONSOLE_LOAD:
72:     case CONSOLE_ALTER:
73:         UnlockKeyboard();
74:         SendBCDTo1415(bcd_key);
75:         break;
76:     case CONSOLE_ADDR:                            // Only digits allowed
77:         if(isdigit(Key)) {
78:             UnlockKeyboard();
79:             SendBCDTo1415(bcd_key);
80:         }
81:         else {
82:             LockKeyboard();
83:         }
84:         break;
85:     }
86: }
87: //-----
88: void __fastcall TFI1415IO::I1415IOKeyDown(TObject *Sender, WORD &Key,
89:                                         TShiftState Shift)
90: {
91:
92:     // Special key handling: Treat Page Down as a Margin Release.
93:
94:     if(Key == VK_NEXT) {
95:         NextLine();
96:     }
97: }
98: //-----
99:
100: // Method to set the state of the console
101:
102: bool __fastcall TFI1415IO::SetState(int s)
103: {
104:     switch(s) {
105:     case CONSOLE_IDLE:
106:         InqReq -> Enabled = true;
107:         InqRlse -> Enabled = false;
108:         InqCancel -> Enabled = false;
109:         WordMark -> Enabled = false;
110:         break;
111:     case CONSOLE_NORMAL:
112:         InqReq -> Enabled = true;
113:         InqRlse -> Enabled = true;
114:         InqCancel -> Enabled = true;
115:         WordMark -> Enabled = false;
116:         break;
117:     case CONSOLE_LOAD:
118:         InqReq -> Enabled = true;
119:         InqRlse -> Enabled = true;
120:         InqCancel -> Enabled = true;
121:         WordMark -> Enabled = true;
122:         break;
123:     case CONSOLE_ALTER:
124:         InqReq -> Enabled = false;
125:         InqRlse -> Enabled = false;
126:         InqCancel -> Enabled = false;
127:         WordMark -> Enabled = true;
128:         break;
129:     case CONSOLE_ADDR:
130:         InqReq -> Enabled = false;
131:         InqRlse -> Enabled = false;
132:         InqCancel -> Enabled = false;
```

```
133:         WordMark -> Enabled = false;
134:         break;
135:     default:
136:         return(false);
137:     }
138:     state = s;
139:     return(true);
140: }
141:
142:
143: //
144: // Utility routine to go to next line on console
145: //
146:
147: void __fastcall TFI1415IO::NextLine()
148: {
149:     I1415IO -> Lines -> Add("");    // Placeholder for wordmarks
150:     I1415IO -> Lines -> Add("");
151: }
152:
153: //
154: // Utility routines to lock/unlock keyboard - display/clear light.
155: //
156:
157: void __fastcall TFI1415IO::LockKeyboard()
158: {
159:     KeyboardLock -> Enabled = true;
160:     KeyboardLockReset -> Enabled = true;
161: }
162:
163: void __fastcall TFI1415IO::UnlockKeyboard()
164: {
165:     KeyboardLock -> Enabled = false;
166:     KeyboardLockReset -> Enabled = false;
167: }
168:
169: //
170: // Utility routine to handle wordmark key
171: //
172:
173: void __fastcall TFI1415IO::DoWordMark()          Ch. 4
174: {
175:     int last;
176:
177:     if(state != CONSOLE_LOAD && state != CONSOLE_ALTER) {
178:         LockKeyboard();
179:         return;
180:     }
181:
182:     last = I1415IO -> Lines -> Count - 1;
183:     if(last <= 0) {
184:         NextLine();
185:         last = I1415IO -> Lines -> Count -1;
186:     }
187:
188:     if(I1415IO -> Lines -> Strings[last-1].Length() <=
189:        I1415IO -> Lines -> Strings[last].Length() ) {
190:         I1415IO -> Lines -> Strings[last-1] =
191:             I1415IO -> Lines -> Strings[last-1] + "v";
192:         UnlockKeyboard();
193:     }
194:     else {
195:         LockKeyboard();
196:     }
197: }
198:
```

```
199: //  
200: // Utility routine to send data to console from  
201: //  
202:  
203: void __fastcall TFI1415IO::SendBCDTo1415(BCD bcd)  
204: {  
205:     int last;  
206:  
207:     last = I1415IO -> Lines -> Count - 1;  
208:     if(last <= 0) {  
209:         NextLine();  
210:         last = I1415IO -> Lines -> Count -1;  
211:     }  
212:  
213:     if(I1415IO -> Lines -> Strings[last-1].Length() <=  
214:         I1415IO -> Lines -> Strings[last].Length()) {  
215:         I1415IO -> Lines -> Strings[last-1] =  
216:             I1415IO -> Lines -> Strings[last-1] + " ";  
217:     }  
218:     I1415IO -> Lines -> Strings[last] =  
219:         I1415IO -> Lines -> Strings[last] + bcd.ToAscii();  
220:     if(I1415IO -> Lines -> Strings[last].Length() > 80) {  
221:         NextLine();  
222:     }  
223:     I1415IO -> Modified = true;  
224: }  
225:  
226: void __fastcall TFI1415IO::WordMarkClick(TObject *Sender)  
227: {  
228:     DoWordMark();  
229:     FocusControl(I1415IO);  
230: }  
231: //-----  
232: void __fastcall TFI1415IO::MarginReleaseClick(TObject *Sender)  
233: {  
234:     NextLine();  
235:     FocusControl(I1415IO);  
236: }  
237: //-----  
238: void __fastcall TFI1415IO::KeyboardLockResetTimer(TObject *Sender)  
239: {  
240:     UnlockKeyboard();  
241: }  
242: //-----
```

```
1: //-----
2: #ifndef UI1415LH
3: #define UI1415LH
4: //-----
5: #include <vcl\Classes.hpp>
6: #include <vcl\Controls.hpp>
7: #include <vcl\StdCtrls.hpp>
8: #include <vcl\Forms.hpp>
9: #include <vcl\ComCtrls.hpp>
10: #include <vcl\ExtCtrls.hpp>
11: //-----
12:
13: // 1415 Light panel display
14:
15: class TF1415L : public TForm
16: {
17:     __published:    // IDE-managed Components
18:     TPageControl *T1415Display;
19:     TTabSheet *TSCPStatus;
20:     TTabSheet *TSIOChannels;
21:     TTabSheet *TSSystemCheck;
22:     TTabSheet *TSPowerSystemControls;
23:     TPanel *PCPU;
24:     TPanel *PStatus;
25:     TLabel *LabelCPU;
26:     TPanel *PIRing;
27:     TLabel *Light_I_OP;
28:     TLabel *Light_I_1;
29:     TLabel *Light_I_2;
30:     TLabel *Light_I_3;
31:     TLabel *Light_I_4;
32:     TLabel *Light_I_5;
33:     TLabel *Light_I_6;
34:     TLabel *Light_I_7;
35:     TLabel *Light_I_8;
36:     TLabel *Light_I_9;
37:     TLabel *Light_I_10;
38:     TLabel *Light_I_11;
39:     TLabel *Light_I_12;
40:     TLabel *LabelIRing;
41:     TPanel *PAring;
42:     TLabel *LabelARing;
43:     TLabel *Light_A_1;
44:     TLabel *Light_A_2;
45:     TLabel *Light_A_3;
46:     TLabel *Light_A_4;
47:     TLabel *Light_A_5;
48:     TLabel *Light_A_6;
49:     TPanel *PClock;
50:     TLabel *LabelClock;
51:     TLabel *Light_Clk_A;
52:     TLabel *Light_Clk_B;
53:     TLabel *Light_Clk_C;
54:     TLabel *Light_Clk_D;
55:     TLabel *Light_Clk_E;
56:     TLabel *Light_Clk_F;
57:     TLabel *Light_Clk_G;
58:     TLabel *Light_Clk_H;
59:     TLabel *Light_Clk_J;
60:     TLabel *Light_Clk_K;
61:     TPanel *PScan;
62:     TLabel *LabelScan;
63:     TLabel *Light_Scan_N;
64:     TLabel *Light_Scan_1;
65:     TLabel *Light_Scan_2;
66:     TLabel *Light_Scan_3;
```

```
67:     TPanel *PSubScan;
68:     TLabel *LabelSubScan;
69:     TLabel *Light_Sub_Scan_U;
70:     TLabel *Light_Sub_Scan_B;
71:     TLabel *Light_Sub_Scan_E;
72:     TLabel *Light_Sub_Scan_MQ;
73:     TPanel *PCycle;
74:     TLabel *LabelCycle;
75:     TLabel *Light_Cycle_A;
76:     TLabel *Light_Cycle_B;
77:     TLabel *Light_Cycle_C;
78:     TLabel *Light_Cycle_D;
79:     TLabel *Light_Cycle_E;
80:     TLabel *Light_Cycle_F;
81:     TLabel *Light_Cycle_I;
82:     TLabel *Light_Cycle_X;
83:     TPanel *PArith;
84:     TLabel *LabelArith;
85:     TLabel *Light_Carry_In;
86:     TLabel *Light_Carry_Out;
87:     TLabel *Light_A_Complement;
88:     TLabel *Light_B_Complement;
89:     TLabel *LabelStatus;
90:     TLabel *Light_B_GT_A;
91:     TLabel *Light_B_EQ_A;
92:     TLabel *Light_B_LT_A;
93:     TLabel *Light_Overflow;
94:     TLabel *Light_Divide_Overflow;
95:     TLabel *Light_Zero_Balance;
96:     TPanel *PIOCh;
97:     TPanel *PIOCh1Status;
98:     TLabel *LabelCh1Status;
99:     TLabel *Light_Ch1_NotReady;
100:    TLabel *Light_Ch1_Busy;
101:    TLabel *Light_Ch1_DataCheck;
102:    TLabel *Light_Ch1_Condition;
103:    TLabel *Light_Ch1_WLRecord;
104:    TLabel *Light_Ch1_NoTransfer;
105:    TPanel *PIOCh1Control;
106:    TLabel *LabelCh1Control;
107:    TLabel *Light_Ch1_Interlock;
108:    TLabel *Light_Ch1_RBControl;
109:    TLabel *Light_Ch1_Read;
110:    TLabel *Light_Ch1_Write;
111:    TLabel *Light_Ch1_Overlap;
112:    TLabel *Light_Ch1_NoOverlap;
113:    TPanel *PIOCh2Control;
114:    TLabel *LabelCh2Control;
115:    TLabel *Light_Ch2_Interlock;
116:    TLabel *Light_Ch2_RBControl;
117:    TLabel *Light_Ch2_Read;
118:    TLabel *Light_Ch2_Write;
119:    TLabel *Light_Ch2_Overlap;
120:    TLabel *Light_Ch2_NoOverlap;
121:    TPanel *PIOChControl;
122:    TPanel *PIOChStatus;
123:    TPanel *PIOCh2Status;
124:    TLabel *LabelCh2Status;
125:    TLabel *Light_Ch2_NotReady;
126:    TLabel *Light_Ch2_Busy;
127:    TLabel *Light_Ch2_DataCheck;
128:    TLabel *Light_Ch2_Condition;
129:    TLabel *Light_Ch2_WLRecord;
130:    TLabel *Light_Ch2_NoTransfer;
131:
132: private: // User declarations
```

```
133: public:      // User declarations
134:     __fastcall TF1415L(TComponent* Owner);
135: };
136: //-----
137: extern TF1415L *F1415L;
138: //-----
139: #endif
```