The Manchester Small Scale Experimental Machine

Lee Wittenberg*

21 June 1998

1 Introduction

This document describes the world's first working stored-program electronic digital computer: the "small scale experimental machine" built at the University of Manchester, which ran its first program on 21 June 1948. We not only describe the machine, but implement a simulator written in Java. Both this document and the program code are generated from a single source file.

The SSEM itself will be represented by a single Java class, which derives from **Thread** to enable the interface to work cleanly with the machine while it is executing:

1 ⟨* 1⟩≡ 3b▷

⟨Package imports 19a⟩;
public class ssem
. . . **extends Thread** {
⟨**ssem**'s members and methods 2a⟩;
}

Defines:
ssem, used in chunks 6–9, 13–18, 26, 35c, 37, 38, and 40.

Uses **Thread**.

The simulator has been sucessfully run under Linux (JDK 1.1.5[1]), and Windows 95 (JDK 1.1.6). Some commercial Java compilers, such as Visual J++, choke on the extra semicolons in the tangled code. Until someone writes a filter that will remove extraneous semicolons from Java code, they must be removed manually with a handy text editor, if you need to use one of these compilers.

Although the interface uses only standard components at their "natural" size, it still appears to require a screen resolution of at least 800×600, and may not be completely visible even at that resolution, depending on how the AWT components interact with the display driver. This will be fixed in future versions (if possible), which will display correctly at 800×600 resolution.

[1]The Linux 1.1.3 JDK seems to have some kind of memory leak that causes the system to run out of memory on long-running programs such as Kilburn's factoring program (section 3).

2 The Machine

All we really need to do is create the machine and its interface. The objects do the rest themselves. Although it's not normally a good idea to allow public data members, the components of the interface and the machine are highly interconnected (as they were in the real SSEM), and the program is much cleaner if we allow these two "global" variables.

For some strange reason, even though the machine's thread is careful to yield control at regular intervals (see below) and all the active threads seem to be at the same priority level, the tube displays never get updated until after the thread suspends (although this *should* happen when the thread yields). Reducing the thread's priority seems to solve this problem fairly cleanly (although, on occasion, the simulator appears to skip steps during automatic operation). When a better method of dealing with this problem is found, we will implement it.

2a ⟨**ssem**'s members and methods 2a⟩≡ (1) 3a ▹

```
public static ssem machine;
public static ssem_Interface iface;
public static void main(String[] args) {
   ⟨Parse command line arguments, args[] 2b⟩;
   ssem.machine ← new ssem();
   ssem.machine.setPriority(Thread.NORM_PRIORITY − 1);
   ssem.iface ← new ssem_Interface();
}
public ssem() {
   ⟨Initialize the SSEM 12c⟩;
   start();
}
```

Defines:
args, used in chunks 2b and 12a.
iface, used in chunks 6–9, 15–18, and 40.
machine, used in chunks 13–18, 26, 35c, 37, and 38.
ssem, used in chunks 6–9, 13–18, 26, 35c, 37, 38, and 40.

Uses *NORM_PRIORITY*, *setPriority*, **ssem_Interface** 19b 21b, *start*, **String**, and **Thread**.

2b ⟨Parse command line arguments, *args*[] 2b⟩≡ (2a)

```
for (int i ← 0; i < args.length; i++) {
   ⟨Parse argument args[i] 12a⟩;
}
```

Uses *args* 2a and *length*.

2.1 Storage: Williams Tubes

The storage hardware for the SSEM consisted of cathode ray tubes later commonly referred to as "Williams Tube" memory (after F. C. Williams, who discovered the "anticipation pulse" effect, which made such memory possible [5]). The SSEM had 3 such tubes, the A (accumulator), C (control), and S (store) tubes, holding 1, 2, and 32 lines, respectively. The lines in the C tube were known as the control instruction (C.I.) and present instruction (P.I.) lines.

3a ⟨**ssem**'s members and methods 2a⟩+≡ (1) ◁2a 5b▷

```
public final static int STORE_SIZE ← 32;
private Williams_Tube a_tube ← new Williams_Tube(1);
private Williams_Tube c_tube ← new Williams_Tube(2);
private Williams_Tube s_tube ← new Williams_Tube(STORE_SIZE);
private final static int CI ← 0;
private final static int PI ← 1;
```

Defines:
- *a_tube*, used in chunks 5b, 7, and 8.
- *CI*, used in chunks 6b, 7a, and 9.
- *c_tube*, used in chunks 5–7 and 9.
- *PI*, used in chunk 9.
- *s_tube*, used in chunks 5–9 and 12d.

Uses **Williams_Tube** 3b.

Each Williams Tube held a series of screen lines. Each line in the tubes of our simulator will initially hold the value 0.

3b ⟨* 1⟩+≡ ◁1 4a▷

```
class Williams_Tube {
  private Line[] lines;
  public Williams_Tube(int n) {
    lines ← new Line[n];
    for (int i ← 0; i < n; i++)
      lines[i] ← new Line(0);
  }
  ⟨Williams_Tube's members and methods 4c⟩;
}
```

Defines:
- *lines*, used in chunks 4c and 5a.
- **Williams_Tube**, used in chunks 3a, 5b, and 31a.

Uses **Line** 4a.

Each screen line held a 32-bit two's complement value. Since Java's **int** class is defined exactly that way, we use an **int** to hold this value.

4a ⟨* 1⟩+≡ ◁3b 13a▷

```
class Line {
  public final static int BITS_PER_LINE ← 32;
  public final static int ALL_ONES ← ~0;
  private int value;
  public Line(int n) {
    value ← n;
  }
  ⟨Line's members and methods 4b⟩;
}
```

Defines:
ALL_ONES, used in chunks 9d and 14a.
BITS_PER_LINE, used in chunks 31, 32, and 34b.
Line, used in chunks 3b, 9d, 14a, 31, 32, and 34b.
value, used in chunk 4b.

Naturally, we need to be able to set and retrieve the values of various lines in various tubes.

4b ⟨**Line**'s members and methods 4b⟩≡ (4a)

```
public int get_Value( ) {
  return value;
}
public void set_Value(int n) {
  value ← n;
}
```

Defines:
get_Value, used in chunk 4c.
set_Value, used in chunks 4c and 5a.

Uses *value* 4a.

4c ⟨**Williams_Tube**'s members and methods 4c⟩≡ (3b) 5a▷

```
public int get_Line_Value(int i) {
  return lines[i].get_Value( );
}
public void set_Line_Value(int i, int n) {
  lines[i].set_Value(n);
}
```

Defines:
get_Line_Value, used in chunks 6–9, 18b, and 32a.
set_Line_Value, used in chunks 6–9, 12d, 16b, and 18b.

Uses *get_Value* 4b, *lines* 3b, and *set_Value* 4b.

We also need to be able to determine the number of lines in a given Williams Tube, and clear the contents of a tube:

5a ⟨**Williams_Tube**'s members and methods 4c⟩+≡ (3b) ◁4c

```
public int get_Num_Lines( ) {
  return lines.length;
}
public void clear_Tube( ) {
  for (int i ← 0; i < lines.length; i++)
    lines[i].set_Value(0);
}
```

Defines:
clear_Tube, used in chunks 15–17.
get_Num_Lines, used in chunk 31a.

Uses *length*, *lines* 3b, and *set_Value* 4b.

The interface will need access to each of the storage tubes.

5b ⟨**ssem**'s members and methods 2a⟩+≡ (1) ◁3a 6a▷

```
public Williams_Tube get_A_Tube( ) {
  return a_tube;
}
public Williams_Tube get_C_Tube( ) {
  return c_tube;
}
public Williams_Tube get_S_Tube( ) {
  return s_tube;
}
```

Defines:
get_A_Tube, used in chunks 16a, 17a, and 26.
get_C_Tube, used in chunks 16a, 17a, and 26.
get_S_Tube, used in chunks 15–18 and 26.

Uses *a_tube* 3a, *c_tube* 3a, *s_tube* 3a, and **Williams_Tube** 3b.

0	5	13	16
Address	Unused[a]	Function	

[a]These bits were reserved for a "storage unit number" [9], providing the (never realized) possibility of extending the SSEM's memory to 8192 words simply by adding more storage tubes.

Figure 1: SSEM Instruction Format

2.2 Instructions[2]

Instructions only used the lower 16 bits of a word. Figure 1 shows the layout of an SSEM instruction, with the least significant bit on the left, as was the custom at Manchester. Bits 13–15 specified the 3-bit function code and bits 0-5 the line in the S tube the instruction operates on. All other bits were unused. It is likely that instructions were limited to 16 bits in anticipation of the 2-instruction per word format that was eventually used in the production Mark I [1].

6a ⟨**ssem**'s members and methods 2a⟩+≡ (1) ◁5b 8c▷

```
public final static int FUNC_BITS ← 3;
public final static int FUNC_MASK ← ~(~0 ≪ FUNC_BITS);
public final static int ADDR_BITS ← 13;
public final static int UNUSED_ADDR_BITS ← 8;
public final static int ADDR_MASK
    ← ~(~0 ≪ (ADDR_BITS − UNUSED_ADDR_BITS));
```

Defines:
ADDR_BITS, used in chunk 10b.
ADDR_MASK, used in chunks 9c, 10b, 16b, and 18b.
FUNC_MASK, used in chunk 10b.

2.2.1 Function Code 0: s, C

Function 0 was what we would call an absolute jump. The contents of the specified line in the store were copied to C.I.

6b ⟨Obey the *op*/*addr* instruction 6b⟩≡ (10b) 7a▷

```
case 0:
  c_tube.set_Line_Value(CI, s_tube.get_Line_Value(addr));
  ssem.iface.update_C_Display(CI);
  break;
```

Uses *addr* 10b, *CI* 3a, *c_tube* 3a, *get_Line_Value* 4c, *iface* 2a, *set_Line_Value* 4c, **ssem** 1 2a, *s_tube* 3a, and *update_C_Display* 23b.

[2]Early computer users generally used the term "order" for what we would call an instruction, but Williams, Kilburn and Tootill [9] consistently use "instruction," so that's what we use here.

2.2.2 Function Code 1: $c + s, C$

Function 1 was what we would call a relative jump. The contents of the specified line in the store were added to C.I.

7a ⟨Obey the *op/addr* instruction 6b⟩+≡ (10b) ◁6b 7b▷

case 1:
  c_tube.set_Line_Value(*CI*,
    c_tube.get_Line_Value(*CI*) + *s_tube.get_Line_Value*(*addr*));
  ssem.*iface.update_C_Display*(*CI*);
  break;

Uses *addr* 10b, *CI* 3a, *c_tube* 3a, *get_Line_Value* 4c, *iface* 2a, *set_Line_Value* 4c, **ssem** 1 2a, *s_tube* 3a, and *update_C_Display* 23b.

2.2.3 Function Code 2: $-s, A$

Function 2 was what we might call "load negative." The two's complement of the contents of the specified line in the store was copied to the accumulator.

7b ⟨Obey the *op/addr* instruction 6b⟩+≡ (10b) ◁7a 7c▷

case 2:
  a_tube.set_Line_Value(0, −*s_tube.get_Line_Value*(*addr*));
  ssem.*iface.update_A_Display*(0);
  break;

Uses *addr* 10b, *a_tube* 3a, *get_Line_Value* 4c, *iface* 2a, *set_Line_Value* 4c, **ssem** 1 2a, *s_tube* 3a, and *update_A_Display* 23b.

2.2.4 Function Code 3: a, S

Function 3 copied the contents of the accumulator to the specified line in the store.

7c ⟨Obey the *op/addr* instruction 6b⟩+≡ (10b) ◁7b 8a▷

case 3:
  s_tube.set_Line_Value(*addr*, *a_tube.get_Line_Value*(0));
  ssem.*iface.update_S_Display*(*addr*);
  break;

Uses *addr* 10b, *a_tube* 3a, *get_Line_Value* 4c, *iface* 2a, *set_Line_Value* 4c, **ssem** 1 2a, *s_tube* 3a, and *update_S_Display* 23b.

2.2.5 Function Code 4: $a - s, A$

Function 4 subtracted the contents of the specified line from the accumulator, leaving the result in the accumulator.

The circuitry of the SSEM economized on logic elements by only decoding as much of the function code as absolutely necessary [4], so the "undocumented" function code 5 had the same result as 4. This idea continued in the design of the later Manchester machines, but programmers generally avoided using these undefined operations [3].

8a ⟨Obey the *op/addr* instruction 6b⟩+≡ (10b) ◁7c 8b▷

```
case 4:
case 5:
  a_tube.set_Line_Value(0,
      a_tube.get_Line_Value(0) − s_tube.get_Line_Value(addr));
  ssem.iface.update_A_Display(0);
  break;
```

Uses *addr* 10b, *a_tube* 3a, *get_Line_Value* 4c, *iface* 2a, *set_Line_Value* 4c, **ssem** 1 2a, *s_tube* 3a, and *update_A_Display* 23b.

2.2.6 Function Code 6: TEST

Function 6 determined if the contents of the accumulator were negative. If it was, a flip-flop was set to skip the next instruction.

8b ⟨Obey the *op/addr* instruction 6b⟩+≡ (10b) ◁8a 8d▷

```
case 6:
  if (a_tube.get_Line_Value(0) < 0) {
    test_flip_flop ← true;
  }
  break;
```

Uses *a_tube* 3a, *get_Line_Value* 4c, and *test_flip_flop* 8c.

8c ⟨**ssem**'s members and methods 2a⟩+≡ (1) ◁6a 10a▷

```
  private boolean test_flip_flop ← false;
```

Defines:
test_flip_flop, used in chunks 8b and 9b.

2.2.7 Function Code 7: STOP

Function 7 halted the operation of the machine, and illuminated the stop lamp.

8d ⟨Obey the *op/addr* instruction 6b⟩+≡ (10b) ◁8b

```
case 7:
  set_Prepulse(false);
  ⟨Illuminate the stop lamp 40c⟩;
  break;
```

Uses *set_Prepulse* 10d.

2.3 Executing Instructions

The "rhythm" of the machine was 4 "beats" to the "bar" [9]:

1. Increment C.I.
2. Present instruction to P.I.
3. P.I. to staticisor
4. P.I. obeyed

Lavington [4] states that C.I. was incremented *after* each instruction rather than before. This is incorrect: C.I. was incremented during the first beat of each bar.

9a ⟨Execute a single instruction 9a⟩≡ (10c)
  ⟨Increment C.I. 9b⟩;
  ⟨Move the instruction specified by C.I. to P.I. 9c⟩;
  ⟨Move the contents of P.I. to the staticisor 9d⟩;
  ⟨Obey the instruction in the staticisor 10b⟩;

The first beat of each bar added either +1 or +2 to C.I., depending on the result of a previous TEST instruction, stored in a flip-flop (*test_flip_flop*), which was immediately reset.

9b ⟨Increment C.I. 9b⟩≡ (9a)
```
{
  int n ← test_flip_flop ? +2 : +1;
  c_tube.set_Line_Value(CI, c_tube.get_Line_Value(CI) + n);
  test_flip_flop ← false;
  ssem.iface.update_C_Display(CI);
}
```
Uses *CI* 3a, *c_tube* 3a, *get_Line_Value* 4c, *iface* 2a, *set_Line_Value* 4c, **ssem** 1 2a, *test_flip_flop* 8c, and *update_C_Display* 23b.

We mask off the address bits in C.I. to get the address of the instruction to place in P.I.

9c ⟨Move the instruction specified by C.I. to P.I. 9c⟩≡ (9a)
```
c_tube.set_Line_Value(PI,
    s_tube.get_Line_Value(c_tube.get_Line_Value(CI) & ADDR_MASK));
ssem.iface.update_C_Display(PI);
```
Uses *ADDR_MASK* 6a, *CI* 3a, *c_tube* 3a, *get_Line_Value* 4c, *iface* 2a, *PI* 3a, *set_Line_Value* 4c, **ssem** 1 2a, *s_tube* 3a, and *update_C_Display* 23b.

The instruction in P.I. was not moved directly to the staticisor without modification. Its value flowed through gates opened or closed by the staticisor switches S_0–S_{15} [9]. This was useful for manual operation when a stream of ones was used instead of the contents of P.I., allowing the user to set up instructions on these switches to be executed directly. All these switches must be in the "on" position for automatic operation.

9d ⟨Move the contents of P.I. to the staticisor 9d⟩≡ (9a)
```
staticisor ← get_Manual( ) ? Line.ALL_ONES : c_tube.get_Line_Value(PI);
staticisor &← get_Stat_Switches( );
```
Uses *ALL_ONES* 4a, *c_tube* 3a, *get_Line_Value* 4c, *get_Manual* 12e, *get_Stat_Switches* 14a, **Line** 4a, *PI* 3a, and *staticisor* 10a.

10a ⟨**ssem**'s members and methods 2a⟩+≡ (1) ◁8c 10c▷

```
private int staticisor ← 0;
```

Defines:
staticisor, used in chunks 9d and 10b.

We extract a function code, *op*, and an address, *addr*, from the staticisor, and perform the appropriate operation.

10b ⟨Obey the instruction in the staticisor 10b⟩≡ (9a)

```
{
  int op ← (staticisor ≫ ADDR_BITS) & FUNC_MASK;
  int addr ← staticisor & ADDR_MASK;
  switch (op) {
    ⟨Obey the op/addr instruction 6b⟩;
  }
}
```

Defines:
addr, used in chunks 6–8.

Uses *ADDR_BITS* 6a, *ADDR_MASK* 6a, *FUNC_MASK* 6a, and *staticisor* 10a.

2.4 The Execution Loop

The main work of the simulator is done in an execution loop, which executes single instructions governed by the pre-pulse. Since the machine is initially quiescent, the first thing we do is suspend the thread. After each instruction, we politely allow other threads to execute.

10c ⟨**ssem**'s members and methods 2a⟩+≡ (1) ◁10a 10d▷

```
private boolean pre_pulse ← false;
public void run( ) {
  suspend( );
  while (true) {
    do {
      ⟨Execute a single instruction 9a⟩;
      yield( );
    } while (pre_pulse);
    suspend( );
  }
}
```

Defines:
pre_pulse, used in chunk 10d.

Uses *suspend* and *yield*.

Since *pre_pulse* must be accessible to concurrent threads in order to stop the loop, we must synchronize access to it:

10d ⟨**ssem**'s members and methods 2a⟩+≡ (1) ◁10c 12b▷

```
synchronized public void set_Prepulse(boolean b) {
  pre_pulse ← b;
}
```

Defines:
set_Prepulse, used in chunks 8d, 15, and 38a.

Uses *pre_pulse* 10c.

3 The Factorization Program

One of the first programs to run on the SSEM was one written by Tom Kilburn to determine the highest proper factor of 2^{18} by trying every number from $2^{18}-1$ down, using repeated subtraction instead of division [4, 8]. When this program first ran, it produced the correct answer after 52 minutes. Evidently, it was designed to determine if the machine could run reliably for relatively long periods of time.

The original program has been lost, but a later version (dated 18 July 1948 by G. C. Tootill) has survived, and is reproduced in the following table:

Line	Binary[a]	Operation[b]/Data
0	—	unused[c]
1	0001100000000010—	$-24, A$
2	0101100000000110—	$a, 26$
3	0101100000000010—	$-26, A$
4	1101100000000110—	$a, 27$
5	1110100000000010—	$-23, A$
6	1101100000000001—	$a-27, A$
7	0000000000000011—	TEST
8	0010100000000100—	$c+20, C$
9	0101100000000001—	$a-26, A$
10	1001100000000110—	$a, 25$
11	1001100000000010—	$-25, A$
12	0000000000000011—	TEST
13	0000000000000111—	STOP
14	0101100000000010—	$-26, A$
15	1010100000000001—	$a-21, A$
16	1101100000000110—	$a, 27$
17	1101100000000010—	$-27, A$
18	0101100000000110—	$a, 26$
19	0110100000000000—	$22, C$
20	10111111...	-3
21	10000000...	1
22	00100000...	4
23	0000000000000000011...	-2^{18}
24	1111111111111111100...	$2^{18}-1$
25	—	temporary storage
26	—	temporary storage
27	—	result

[a] "Backwards" binary with the least significant bit on the left.
[b] Early Manchester notation.
[c] Skipped by the initial C.I. increment.

and in Java hexadecimal:

11 ⟨factorization program 11⟩≡ (12d)

```
#0, #4018, #601a, #401a, #601b, #4017, #801b, #c000, #2014, #801a,
    #6019, #4019, #c000, #e000, #401a, #8015, #601b, #401b, #601a,
    #0016, −#3, #1, #4, −#40000, #3ffff
```

We allow a `-demo` or `-d` command line option to preload the factorization program into the S tube:

12a ⟨Parse argument *args*[*i*] 12a⟩≡ (2b)

```
if (args[i].equals("-d") ∨ args[i].equals("-demo"))
    load_demo ← true;
```

Uses *args* 2a, *equals*, and *load_demo* 12b.

12b ⟨**ssem**'s members and methods 2a⟩+≡ (1) ◁10d 12e▷

```
private static boolean load_demo ← false;
```

Defines:
load_demo, used in chunk 12.

12c ⟨Initialize the SSEM 12c⟩≡ (2a)

```
if (load_demo) {
    ⟨Load the factorization program 12d⟩;
}
```

Uses *load_demo* 12b.

12d ⟨Load the factorization program 12d⟩≡ (12c)

```
{
    int[] factor_pgm ← ⟨factorization program 11⟩;
    for (int i ← 0; i < factor_pgm.length; i++)
        s_tube.set_Line_Value(i, factor_pgm[i]);
}
```

Uses *length*, *set_Line_Value* 4c, and *s_tube* 3a.

4 Machine Operation

The SSEM could be operated in either manual or automatic mode. The automatic/manual operation switch controlled this mode. We use a flag whose access is synchronized, since its value is set by the interface rather than the machine (in other words, in a separate thread).

12e ⟨**ssem**'s members and methods 2a⟩+≡ (1) ◁12b 14a▷

```
private boolean manual_mode ← true;
synchronized public void set_Manual(boolean b) {
    manual_mode ← b;
}
synchronized public boolean get_Manual( ) {
    return manual_mode;
}
```

Defines:
get_Manual, used in chunk 9d.
set_Manual, used in chunks 13a and 35c.

13a ⟨* 1⟩+≡ ◁4a 13b▷

```
class Manual_Mode_Callback
...extends Callback {
  private boolean val;
  public Manual_Mode_Callback(boolean b) {
    val ← b;
  }
  public void func(AWTEvent e) {
    ssem.machine.set_Manual(val);
  }
}
```

Defines:
func, used in chunk 20.
Manual_Mode_Callback, used in chunk 35c.

Uses **AWTEvent**, **Callback** 19c, *machine* 2a, *set_Manual* 12e, and **ssem** 1 2a.

4.1 Manual Operation

The KSP key sent a single pre-pulse to the SSEM to execute a single instruction. It could also be used during automatic operation to "single-step" through a program. Since we don't actually set the pre-pulse flag here, the execution loop will only cycle once when it is resumed. The pre-pulse turns off the stop lamp.

13b ⟨* 1⟩+≡ ◁13a 14b▷

```
class KSP_Callback
...extends Callback {
  public void func(AWTEvent e) {
    ⟨Turn off the stop lamp 40d⟩;
    ssem.machine.resume( );
  }
}
```

Defines:
func, used in chunk 20.
KSP_Callback, used in chunk 38c.

Uses **AWTEvent**, **Callback** 19c, *machine* 2a, *resume*, and **ssem** 1 2a.

During manual operation, instructions were taken from the S_0–S_{15} switches rather than the store. We synchronize access to our staticisor switches just as we do for the automatic/manual flag, and for the same reasons.

14a ⟨**ssem**'s members and methods 2a⟩+≡ (1) ◁12e 17b▷

```
private int staticisor_switches ← Line.ALL_ONES;
synchronized public int get_Stat_Switches( ) {
  return staticisor_switches;
}
synchronized public void set_Stat_Switch_Bit(int n) {
  staticisor_switches ←| (#1 ≪ n);
}
synchronized public void reset_Stat_Switch_Bit(int n) {
  staticisor_switches ←& ∼(#1 ≪ n);
}
```

Defines:
get_Stat_Switches, used in chunks 9d, 16b, and 18b.
reset_Stat_Switch_Bit, used in chunk 14b.
set_Stat_Switch_Bit, used in chunks 14b and 37a.

Uses *ALL_ONES* 4a and **Line** 4a.

Each staticisor switch sets (or resets) a single bit. 1 indicates a closed switch, 0 an open one.

14b ⟨* 1⟩+≡ ◁13b 15a▷

```
class Stat_Switch_Callback
...extends Callback {
  private int bit;
  private boolean val;
  public Stat_Switch_Callback(int n, boolean b) {
    bit ← n;
    val ← b;
  }
  public void func(AWTEvent e) {
    if (val)
      ssem.machine.set_Stat_Switch_Bit(bit);
    else
      ssem.machine.reset_Stat_Switch_Bit(bit);
  }
}
```

Defines:
func, used in chunk 20.
Stat_Switch_Callback, used in chunk 37a.

Uses **AWTEvent**, **Callback** 19c, *machine* 2a, *reset_Stat_Switch_Bit* 14a, *set_Stat_Switch_Bit* 14a, and **ssem** 1 2a.

4.2 Automatic Operation

Closing the pre-pulse switch started automatic execution, taking instructions from the S tube, and turning off the stop lamp.

15a ⟨* 1⟩+≡ ◁14b 15b▷

```
class Start_Exec_Callback
...extends Callback {
  public void func(AWTEvent e) {
    ssem.machine.set_Prepulse(true);
    ⟨Turn off the stop lamp 40d⟩;
    ssem.machine.resume( );
  }
}
```

Defines:
func, used in chunk 20.
Start_Exec_Callback, used in chunk 38a.

Uses **AWTEvent**, **Callback** 19c, *machine* 2a, *resume*, *set_Prepulse* 10d, and **ssem** 1 2a.

Opening the pre-pulse switch stopped automatic execution, which could be resumed by closing the switch or pressing the KSP key to execute a single instruction.

15b ⟨* 1⟩+≡ ◁15a 15c▷

```
class Stop_Exec_Callback
...extends Callback {
  public void func(AWTEvent e) {
    ssem.machine.set_Prepulse(false);
  }
}
```

Defines:
func, used in chunk 20.
Stop_Exec_Callback, used in chunk 38a.

Uses **AWTEvent**, **Callback** 19c, *machine* 2a, *set_Prepulse* 10d, and **ssem** 1 2a.

4.3 Erasing Information

The KSC key cleared the contents of the S tube.

15c ⟨* 1⟩+≡ ◁15b 16a▷

```
class KSC_Callback
...extends Callback {
  public void func(AWTEvent e) {
    ssem.machine.get_S_Tube( ).clear_Tube( );
    ssem.iface.update_S_Display( );
  }
}
```

Defines:
func, used in chunk 20.
KSC_Callback, used in chunk 39a.

Uses **AWTEvent**, **Callback** 19c, *clear_Tube* 5a, *get_S_Tube* 5b, *iface* 2a, *machine* 2a, **ssem** 1 2a, and *update_S_Display* 23b.

The KCC key cleared the contents of the A and C tubes.

16a ⟨* 1⟩+≡ ◁15c 16b▷
```
class KCC_Callback
...extends Callback {
  public void func(AWTEvent e) {
    ssem.machine.get_A_Tube().clear_Tube();
    ssem.machine.get_C_Tube().clear_Tube();
    ssem.iface.update_A_Display();
    ssem.iface.update_C_Display();
  }
}
```
Defines:
func, used in chunk 20.
KCC_Callback, used in chunk 38e.

Uses **AWTEvent**, **Callback** 19c, *clear_Tube* 5a, *get_A_Tube* 5b, *get_C_Tube* 5b, *iface* 2a, *machine* 2a, **ssem** 1 2a, *update_A_Display* 23b, and *update_C_Display* 23b.

The KLC key cleared the contents of a single line (specified by the staticisor switches) of the S tube. Actually, it appears that if the key was pressed during automatic operation, things got a bit complicated, involving the contents of P.I., as well. However, this doesn't appear to be very useful, so we're not going to bother for now.

16b ⟨* 1⟩+≡ ◁16a 17a▷
```
class KLC_Callback
...extends Callback {
  public void func(AWTEvent e) {
    int n ← ssem.machine.get_Stat_Switches() & ssem.ADDR_MASK;
    ssem.machine.get_S_Tube().set_Line_Value(n, 0);
    ssem.iface.update_S_Display(n);
  }
}
```
Defines:
func, used in chunk 20.
KLC_Callback, used in chunk 38d.

Uses *ADDR_MASK* 6a, **AWTEvent**, **Callback** 19c, *get_Stat_Switches* 14a, *get_S_Tube* 5b, *iface* 2a, *machine* 2a, *set_Line_Value* 4c, **ssem** 1 2a, and *update_S_Display* 23b.

The KAC and KEC keys were not operational in June 1948, but they appear to have existed anyway [2, 6], and seem useful, so we'll provide them for our simulator. The KAC key cleared the A Tube only; the KEC key cleared all 3 tubes.

17a ⟨* 1⟩+≡ ◃16b 18a▹

```
class KAC_Callback
...extends Callback {
  public void func(AWTEvent e) {
    ssem.machine.get_A_Tube().clear_Tube();
    ssem.iface.update_A_Display();
  }
}
class KEC_Callback
...extends Callback {
  public void func(AWTEvent e) {
    ssem.machine.get_A_Tube().clear_Tube();
    ssem.machine.get_C_Tube().clear_Tube();
    ssem.machine.get_S_Tube().clear_Tube();
    ssem.iface.update_A_Display();
    ssem.iface.update_C_Display();
    ssem.iface.update_S_Display();
  }
}
```

Defines:
func, used in chunk 20.
KAC_Callback, used in chunk 39b.
KEC_Callback, used in chunk 39c.

Uses **AWTEvent**, **Callback** 19c, *clear_Tube* 5a, *get_A_Tube* 5b, *get_C_Tube* 5b, *get_S_Tube* 5b, *iface* 2a, *machine* 2a, **ssem** 1 2a, *update_A_Display* 23b, *update_C_Display* 23b, and *update_S_Display* 23b.

4.4 Setting Up the Store

The write/erase switch determined whether the typewriter buttons wrote (set) or erased (reset) bits on the specified line in the S tube.

17b ⟨**ssem**'s members and methods 2a⟩+≡ (1) ◃14a

```
private boolean write_mode ← true;
public void set_Write_Flag(boolean b) {
  write_mode ← b;
}
public boolean get_Write_Flag() {
  return write_mode;
}
```

Defines:
get_Write_Flag, used in chunk 18b.
set_Write_Flag, used in chunks 18a and 38b.

18a ⟨* 1⟩+≡ ◁17a 18b▷

```
class Write_Flag_Callback
...extends Callback {
  private boolean val;
  public Write_Flag_Callback(boolean b) {
    val ← b;
  }
  public void func(AWTEvent e) {
    ssem.machine.set_Write_Flag(val);
  }
}
```

Defines:
func, used in chunk 20.
Write_Flag_Callback, used in chunk 38b.

Uses **AWTEvent**, **Callback** 19c, *machine* 2a, *set_Write_Flag* 17b, and **ssem** 1 2a.

Each of the typewriter buttons was used to set (or reset, depending on the setting of the write/erase switch) a particular bit in the line in the S Tube specified by the staticisor switches.

18b ⟨* 1⟩+≡ ◁18a 19b▷

```
class Typewriter_Callback
...extends Callback {
  private int mask;
  public Typewriter_Callback(int n) {
    mask ← #1 ≪ n;
  }
  public void func(AWTEvent e) {
    int i ← ssem.machine.get_Stat_Switches( ) & ssem.ADDR_MASK;
    int n ← ssem.machine.get_S_Tube( ).get_Line_Value(i);
    if (ssem.machine.get_Write_Flag( ))
      n |← mask;
    else
      n &← ∼mask;
    ssem.machine.get_S_Tube( ).set_Line_Value(i, n);
    ssem.iface.update_S_Display(i);
  }
}
```

Defines:
func, used in chunk 20.
Typewriter_Callback, used in chunk 34b.

Uses *ADDR_MASK* 6a, **AWTEvent**, **Callback** 19c, *get_Line_Value* 4c, *get_Stat_Switches* 14a, *get_S_Tube* 5b, *get_Write_Flag* 17b, *iface* 2a, *machine* 2a, *set_Line_Value* 4c, **ssem** 1 2a, and *update_S_Display* 23b.

5 The User Interface

To make our interface intelligible to our users, we use the modern GUI idiom of menus, buttons, and checkboxes. For portability, we use only standard Java AWT[3] components. In future, we may decide to provide a more realistic interface, utilizing bitmap pictures of the actual machine, but this seems best for now.

19a ⟨Package imports 19a⟩≡ (1) 21e▷

```
import java.awt.*;
```

Uses *awt* and *java*.

The interface itself will be a frame. It would be nice if it were also an applet, but that will have to wait for another day.

19b ⟨* 1⟩+≡ ◁18b 19c▷

```
class ssem_Interface
...extends Frame {
  ⟨ssem_Interface's members and methods 21b⟩;
}
```

Defines:
ssem_Interface, used in chunk 2a.

Uses **Frame**.

5.1 Callbacks

Since we find the idea of "callback" functions more intelligible than that of "listeners," we create a **Callback** class to deal with this.

19c ⟨* 1⟩+≡ ◁19b 19d▷

```
abstract class Callback {
  public abstract void func(AWTEvent e);
}
```

Defines:
Callback, used in chunks 13–21 and 27a.
func, used in chunk 20.

Uses **AWTEvent**.

5.1.1 Buttons

When a button is pressed, it generates an "action" event. We define a class derived from **Button**, containing a callback,

19d ⟨* 1⟩+≡ ◁19c 20c▷

```
class CB_Button
...extends Button
...implements ActionListener {
  private Callback cb;
  ⟨CB_Button's members and methods 20a⟩;
}
```

Defines:
cb, used in chunks 20 and 21a.
CB_Button, used in chunks 34b, 38, and 39.

Uses **ActionListener**, **Button**, and **Callback** 19c.

[3] Version 1.1.x at this writing.

whose *func* will be executed by the button's *actionPerformed* method.

20a ⟨**CB_Button**'s members and methods 20a⟩≡ (19d) 20b▷

```
public void actionPerformed(ActionEvent e) {
  cb.func(e);
}
```

Uses **ActionEvent**, *cb* 19d 20c, and *func* 13a 13b 14b 15a 15b 15c 16a 16b 17a 18a 18b 19c 27a.

A callback button is created by supplying a string for the button's label and a callback for its action, then registering it as the listener for this button.

20b ⟨**CB_Button**'s members and methods 20a⟩+≡ (19d) ◁20a

```
public CB_Button(String s, Callback c) {
  super(s);
  cb ← c;
  addActionListener(this);
}
```

Defines:
CB_Button, used in chunks 34b, 38, and 39.
Uses *addActionListener*, **Callback** 19c, *cb* 19d 20c, and **String**.

5.1.2 Checkboxes

Checkboxes generate "item" events when selected. As with **CB_Button**, we derive a class from **Checkbox:**

20c ⟨* 1⟩+≡ ◁19d 26▷

```
class CB_Checkbox
...extends Checkbox
...implements ItemListener {
  private Callback cb;
  ⟨CB_Checkbox's members and methods 20d⟩;
}
```

Defines:
cb, used in chunks 20 and 21a.
CB_Checkbox, used in chunks 26, 35c, 37, and 38.
Uses **Callback** 19c, **Checkbox**, and **ItemListener**.

whose *func* will be executed by *itemStateChanged*.

20d ⟨**CB_Checkbox**'s members and methods 20d⟩≡ (20c) 20e▷

```
public void itemStateChanged(ItemEvent e) {
  cb.func(e);
}
```

Uses *cb* 19d 20c, *func* 13a 13b 14b 15a 15b 15c 16a 16b 17a 18a 18b 19c 27a, and **ItemEvent**.

A callback checkbox is created pretty much the same way as a callback button, except that we allow for an initial setting,

20e ⟨**CB_Checkbox**'s members and methods 20d⟩+≡ (20c) ◁20d 21a▷

```
public CB_Checkbox(boolean b, Callback c) {
  super("", Λ, b);
  cb ← c;
  addItemListener(this);
}
```

Defines:
CB_Checkbox, used in chunks 26, 35c, 37, and 38.
Uses *addItemListener*, **Callback** 19c, and *cb* 19d 20c.

and a second constructor is necessary, since we also use checkbox groups to implement "radio buttons."

21a ⟨**CB_Checkbox**'s members and methods 20d⟩+≡ (20c) ◁20e

```
public CB_Checkbox(String s, CheckboxGroup g, boolean b,
        Callback c) {
    super(s, g, b);
    cb ← c;
    addItemListener(this);
}
```

Defines:
CB_Checkbox, used in chunks 26, 35c, 37, and 38.

Uses *addItemListener*, **Callback** 19c, *cb* 19d 20c, **CheckboxGroup**, and **String**.

5.2 Building the Interface

The interface is built by the **ssem_Interface** constructor. After building the interface, we adjust its size (*pack*), prevent it from being resized (*setResizable*), and display it (*setVisible*).

21b ⟨**ssem_Interface**'s members and methods 21b⟩≡ (19b) 21d▷

```
public ssem_Interface( ) {
    setTitle("SSEM␣Simulator");
    ⟨Allow the window to be closed 21c⟩;
    ⟨Build the interface 22⟩;
    pack( );
    setResizable(false);
    setVisible(true);
}
```

Defines:
ssem_Interface, used in chunk 2a.

Uses *pack*, *setResizable*, *setTitle*, and *setVisible*.

We use an inner class to extend **WindowAdapter** and exit the program when the window is closed.

21c ⟨Allow the window to be closed 21c⟩≡ (21b)

```
addWindowListener(new ProgramCloser( ));
```

Uses *addWindowListener* and **ProgramCloser** 21d.

21d ⟨**ssem_Interface**'s members and methods 21b⟩+≡ (19b) ◁21b 23a▷

```
private class ProgramCloser
...extends WindowAdapter {
    public void windowClosing(WindowEvent e) {
        System.exit(0);
    }
}
```

Defines:
ProgramCloser, used in chunk 21c.

Uses *exit*, **System**, **WindowAdapter**, and **WindowEvent**.

21e ⟨Package imports 19a⟩+≡ (1) ◁19a

```
import java.awt.event.*;
```

Uses *awt*, *event*, and *java*.

The interface consists of 4 main components: the display panel containing the A, C and S tube displays; the typewriter panel containing the "typewriter" buttons; the control panel containing the control keys and switches; and the staticisor panel containing the staticisor switches. The display panel is at the top, and the control panel on the bottom.

Since the contents of the displays in the display panel change as the machine executes, we need to be able to access them. The data members *a_display*, *c_display* and *s_display* will make this access possible.

Unfortunately, we can't use a simple grid layout since it would expand each panel to the same height. In order to get a single column with rows of differing height, we need to use a grid bag layout, but the *gridx*, *gridwidth*, and *gridheight* constraints will be the same for all of the components. Only *gridy* varies.

Also, because the display panel cannot be set up using a layout manager (see Section 5.2.1), we have to pack the components and then adjust the spacing of the display panel's components before we're done.

22 ⟨Build the interface 22⟩≡ (21b)

```
{
  ⟨Set up a grid bag layout, gbl, with constraints gbc 42a⟩;
  gbc.gridx ← 0;
  gbc.gridwidth ← gbc.gridheight ← 1;
  Component x;
  Display_Panel d;
  x ← d ← new Display_Panel( );
  a_display ← d.get_A_Display( );
  c_display ← d.get_C_Display( );
  s_display ← d.get_S_Display( );
  gbc.gridy ← 0;
  ⟨Add component x to grid bag gbl, with constraints gbc 42b⟩;
  x ← new Typewriter_Panel( );
  gbc.gridy ← 1;
  ⟨Add component x to grid bag gbl, with constraints gbc 42b⟩;
  x ← new Stat_Panel( );
  gbc.gridy ← 2;
  ⟨Add component x to grid bag gbl, with constraints gbc 42b⟩;
  Control_Panel p;
  x ← p ← new Control_Panel( );
  lamp ← p.get_Lamp( );
  gbc.gridy ← 3;
  ⟨Add component x to grid bag gbl, with constraints gbc 42b⟩;
  pack( );
  d.adjust( );
}
```

Uses *a_display* 23b, *adjust* 26, *c_display* 23b, **Component**, **Control_Panel** 37b, **Display_Panel** 26, *gbc* 42a, *get_A_Display* 25, *get_C_Display* 25, *get_Lamp* 23a, *get_S_Display* 25, *gridheight*, *gridwidth*, *gridx*, *gridy*, *lamp* 23a 39e, *pack*, *s_display* 23b, **Stat_Panel** 35a 35b, and **Typewriter_Panel** 34a.

23a ⟨**ssem_Interface**'s members and methods 21b⟩+≡ (19b) ◁21d 23b▷

```
private Stop_Lamp lamp;
public Stop_Lamp get_Lamp( ) {
  return lamp;
}
```

Defines:
get_Lamp, used in chunks 22, 39, and 40.
lamp, used in chunks 22 and 39d.

Uses **Stop_Lamp** 39e 39f.

The only access other elements of the machine need to the various displays is to be able to update them, so we supply *update_A_Display*, *update_C_Display* and *update_S_Display*. Often only one line of any display needs to be updated at any given time, hence the versions with a parameter n to specify the line to be updated.

23b ⟨**ssem_Interface**'s members and methods 21b⟩+≡ (19b) ◁23a

```
private Display_Tube a_display;
private Display_Tube c_display;
private Display_Tube s_display;
public void update_A_Display( ) {
  a_display.repaint( );
}
public void update_A_Display(int n) {
  a_display.repaint(n);
}
public void update_C_Display( ) {
  c_display.repaint( );
}
public void update_C_Display(int n) {
  c_display.repaint(n);
}
public void update_S_Display( ) {
  s_display.repaint( );
}
public void update_S_Display(int n) {
  s_display.repaint(n);
}
```

Defines:
a_display, used in chunk 22.
c_display, used in chunk 22.
s_display, used in chunk 22.
update_A_Display, used in chunks 7b, 8a, 16a, and 17a.
update_C_Display, used in chunks 6b, 7a, 9, 16a, and 17a.
update_S_Display, used in chunks 7c and 15–18.

Uses **Display_Tube** 30b 31a and *repaint* 32c.

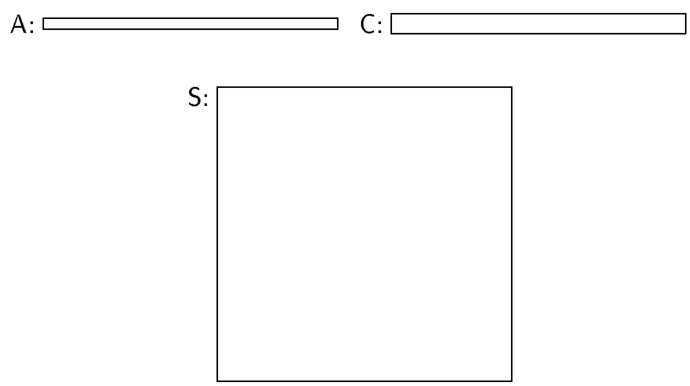

Figure 2: The Display Panel

5.2.1 The Display Panel

The display panel contains the displays for the 3 Williams Tube memories, the A, C, and S tubes. The panel is laid out as shown in Figure 2.

Each of the display tubes has a checkbox that the user can use to disable animation of that particular tube. Disabling tube animation can significantly speed up execution of the simulator. This feature was shamelessly stolen from Martin Campbell-Kelly's EDSAC simulators.

24 ⟨Keep a list, *parts*, of display panel components 24⟩≡ (26)

private Component[ ] *parts*;
private final static int $A\_LABEL \leftarrow 0$;
private final static int $A\_BOX \leftarrow 1$;
private final static int $A\_TUBE \leftarrow 2$;
private final static int $C\_LABEL \leftarrow 3$;
private final static int $C\_BOX \leftarrow 4$;
private final static int $C\_TUBE \leftarrow 5$;
private final static int $S\_LABEL \leftarrow 6$;
private final static int $S\_BOX \leftarrow 7$;
private final static int $S\_TUBE \leftarrow 8$;
private final static int $NUM\_PARTS \leftarrow S\_TUBE + 1$;

Defines:
A_BOX, used in chunks 26 and 29a.
A_LABEL, used in chunks 26, 28a, and 29a.
A_TUBE, used in chunks 25, 26, and 29a.
C_BOX, used in chunks 26 and 29b.
C_LABEL, used in chunks 26 and 29b.
C_TUBE, used in chunks 25, 26, 28a, and 29b.
NUM_PARTS, used in chunk 26.
parts, used in chunks 25, 26, and 28–30.
S_BOX, used in chunks 26 and 30a.
S_LABEL, used in chunks 26 and 30a.
S_TUBE, used in chunks 25, 26, and 30a.

Uses **Component**.

Each display tube needs to be accessed directly for updating during program execution.

25 ⟨**Display_Panel**'s members and methods 25⟩≡ (26)

```
public Display_Tube get_A_Display( ) {
  return (Display_Tube)parts[A_TUBE];
}
public Display_Tube get_C_Display( ) {
  return (Display_Tube)parts[C_TUBE];
}
public Display_Tube get_S_Display( ) {
  return (Display_Tube)parts[S_TUBE];
}
```

Defines:
get_A_Display, used in chunk 22.
get_C_Display, used in chunk 22.
get_S_Display, used in chunk 22.

Uses *A_TUBE* 24, *C_TUBE* 24, **Display_Tube** 30b 31a, *parts* 24, and *S_TUBE* 24.

We want the S tube display to be centered in the panel, which means that none of the AWT's standard layout managers will do the job. We will have to place each component explicitly. Given the preferred size of each component, we can easily calculate the proper size for the panel and the proper location for each component. Unfortunately, until the frame containing them is either packed or shown, AWT components report their preferred size as 0×0. We get around this problem by not bothering to do any layout when the panel is created, but provide the *adjust* method, invoked only after the frame has been packed, to lay out the components properly.

We create the display tubes first, so that the checkbox callbacks can reference them directly to disable/enable the appropriate display. Each display will initially be enabled, so these checkboxes should be on.

26 ⟨* 1⟩+≡ ◁20c 27a▷

```
class Display_Panel
...extends Panel {
  ⟨Keep a list, parts, of display panel components 24⟩;
  public Display_Panel( ) {
    setLayout(Λ);
    parts ← new Component[NUM_PARTS];
    parts[A_TUBE] ← new Display_Tube(ssem.machine.get_A_Tube( ));
    parts[C_TUBE] ← new Display_Tube(ssem.machine.get_C_Tube( ));
    parts[S_TUBE] ← new Display_Tube(ssem.machine.get_S_Tube( ));
    add(parts[A_LABEL] ← new Label("A:"));
    add(parts[A_BOX] ← new CB_Checkbox(true,
        new Tube_Show_Callback(parts[A_TUBE])));
    add(parts[A_TUBE]);
    add(parts[C_LABEL] ← new Label("C:"));
    add(parts[C_BOX] ← new CB_Checkbox(true,
        new Tube_Show_Callback(parts[C_TUBE])));
    add(parts[C_TUBE]);
    add(parts[S_LABEL] ← new Label("S:"));
    add(parts[S_BOX] ← new CB_Checkbox(true,
        new Tube_Show_Callback(parts[S_TUBE])));
    add(parts[S_TUBE]);
  }
  public void adjust( ) {
    ⟨Adjust the layout of the current display panel 27b⟩;
  }
  ⟨Display_Panel's members and methods 25⟩;
}
```

Defines:
adjust, used in chunk 22.
Display_Panel, used in chunk 22.

Uses *A_BOX* 24, *add*, *A_LABEL* 24, *A_TUBE* 24, **CB_Checkbox** 20c 20e 21a, *C_BOX* 24, *C_LABEL* 24, **Component**, *C_TUBE* 24, **Display_Tube** 30b 31a, *get_A_Tube* 5b, *get_C_Tube* 5b, *get_S_Tube* 5b, **Label**, *machine* 2a, *NUM_PARTS* 24, **Panel**, *parts* 24, *S_BOX* 24, *setLayout*, *S_LABEL* 24, **ssem** 1 2a, *S_TUBE* 24, and **Tube_Show_Callback** 27a.

Each checkbox will simply enable or disable its specified display tube.

27a ⟨* 1⟩+≡ ◁26 30b▷

```
class Tube_Show_Callback
...extends Callback {
  private Display_Tube tube;
  public Tube_Show_Callback(Component t) {
    tube ← (Display_Tube)t;
  }
  public void func(AWTEvent e) {
    tube.setEnabled(((Checkbox)e.getSource()).getState());
    tube.repaint();
  }
}
```

Defines:
func, used in chunk 20.
Tube_Show_Callback, used in chunk 26.

Uses **AWTEvent**, **Callback** 19c, **Checkbox**, **Component**, **Display_Tube** 30b 31a, *getSource*, *getState*, *repaint* 32c, and *setEnabled* 33b.

To achieve the layout shown in Figure 2, we must determine the total height and width needed by the panel. We do this by first computing the width and height (*p_width* and *p_height*, respectively) of the top part (containing the A and C displays) of the panel. Since the bottom part (containing the S display) is necessarily narrower than the top, this is the actual width of the panel. We use the preliminary height value when laying out the A and C tubes to align each component along a centered horizontal axis. We then complete the actual height calculation as we lay out the S tube display.

We will leave horizontal and vertical margins of 8 pixels around the components in each panel part, and an extra gap of 10 times that much between the A and C tubes.

27b ⟨Adjust the layout of the current display panel 27b⟩≡ (26)

```
{
  int p_width, p_height;
  int margin ← 8, extra_gap ← 10 × margin;
  ⟨Compute p_width and p_height for the top part of the panel 28a⟩;
  ⟨Lay out the A and C tube displays 28b⟩;
  ⟨Lay out the S tube display, adjusting p_height appropriately 30a⟩;
  setSize(p_width, p_height);
}
```

Defines:
extra_gap, used in chunk 28.
margin, used in chunks 28 and 30a.
p_height, used in chunks 28–30.
p_width, used in chunks 28a and 30a.

Uses *setSize*.

The width of the panel's top part is the sum of the widths of the A and C components plus space for the gap between them and the margins. The height is that of the tallest component, plus space for the margins.

28a ⟨Compute *p_width* and *p_height* for the top part of the panel 28a⟩≡ (27b)

$$
\begin{array}{l}
\{ \\
\quad p\_width \leftarrow p\_height \leftarrow margin; \\
\quad \textbf{for}\ (\textbf{int}\ i \leftarrow A\_LABEL;\ i \le C\_TUBE;\ i\texttt{++})\ \{ \\
\quad\quad \textbf{Dimension}\ d \leftarrow parts[i].getPreferredSize(\,); \\
\quad\quad p\_width \stackrel{+}{\leftarrow} d.width; \\
\quad\quad p\_height \leftarrow \textbf{Math}.max(p\_height,\ d.height); \\
\quad \} \\
\quad p\_width \stackrel{+}{\leftarrow} margin + extra\_gap; \\
\quad p\_height \stackrel{+}{\leftarrow} margin; \\
\}
\end{array}
$$

Uses *A_LABEL* 24, *C_TUBE* 24, **Dimension**, *extra_gap* 27b, *getPreferredSize* 41c, *height*, *margin* 27b, **Math**, *max*, *parts* 24, *p_height* 27b, *p_width* 27b, and *width*.

We use *x_pos* to keep track of the current horizontal position as we lay out the tube displays from left to right, with at gap of *extra_gap* pixels between them.

28b ⟨Lay out the A and C tube displays 28b⟩≡ (27b)

$$
\begin{array}{l}
\{ \\
\quad \textbf{int}\ x\_pos \leftarrow margin; \\
\quad \langle \text{Lay out the A tube display, adjusting } xpos \text{ 29a} \rangle; \\
\quad x\_pos \stackrel{+}{\leftarrow} extra\_gap; \\
\quad \langle \text{Lay out the C tube display, adjusting } xpos \text{ 29b} \rangle; \\
\}
\end{array}
$$

Defines:
x_pos, used in chunk 29.

Uses *extra_gap* 27b and *margin* 27b.

For each component in the A tube display, we use *getPreferredSize* to find its dimensions, d. The current horizontal position is in *x_pos*. Since *p_width* and *p_height* currently hold the dimensions of the upper part of the panel,

$$y\_pos = \frac{p\_height - d.height + 1}{2}$$

will center the component vertically in the top part of the panel.

As we lay out each component, we increment *x_pos* by its width so as to be ready to place the next one.

29a ⟨Lay out the A tube display, adjusting *xpos* 29a⟩≡ (28b)

```
{
  Dimension d;
  d ← parts[A_LABEL].getPreferredSize( );
  parts[A_LABEL].setBounds(x_pos, (p_height − d.height + 1) ÷ 2, d.width,
      d.height);
  x_pos +← d.width;
  d ← parts[A_BOX].getPreferredSize( );
  parts[A_BOX].setBounds(x_pos, (p_height − d.height + 1) ÷ 2, d.width,
      d.height);
  x_pos +← d.width;
  d ← parts[A_TUBE].getPreferredSize( );
  parts[A_TUBE].setBounds(x_pos, (p_height − d.height + 1) ÷ 2, d.width,
      d.height);
  x_pos +← d.width;
}
```

Uses *A_BOX* 24, *A_LABEL* 24, *A_TUBE* 24, **Dimension**, *getPreferredSize* 41c, *height*, *parts* 24, *p_height* 27b, *setBounds*, *width*, and *x_pos* 28b 30a.

The C tube display is laid out exactly the same way as the A tube display.

29b ⟨Lay out the C tube display, adjusting *xpos* 29b⟩≡ (28b)

```
{
  Dimension d;
  d ← parts[C_LABEL].getPreferredSize( );
  parts[C_LABEL].setBounds(x_pos, (p_height − d.height + 1) ÷ 2, d.width,
      d.height);
  x_pos +← d.width;
  d ← parts[C_BOX].getPreferredSize( );
  parts[C_BOX].setBounds(x_pos, (p_height − d.height + 1) ÷ 2, d.width,
      d.height);
  x_pos +← d.width;
  d ← parts[C_TUBE].getPreferredSize( );
  parts[C_TUBE].setBounds(x_pos, (p_height − d.height + 1) ÷ 2, d.width,
      d.height);
  x_pos +← d.width;
}
```

Uses *C_BOX* 24, *C_LABEL* 24, *C_TUBE* 24, **Dimension**, *getPreferredSize* 41c, *height*, *parts* 24, *p_height* 27b, *setBounds*, *width*, and *x_pos* 28b 30a.

The S tube display is a bit trickier, because the pieces are laid out relative to the tube, rather than left-to-right. The tube is centered in the field, the label is laid to the left of where the checkbox should go (aligned at the top of the tube), and the label's height is used to align the checkbox with it, as we did for the A and C tube display components. The tube itself is the tallest component, so we add its height (with margins) to *p_height* to get the true panel height.

30a ⟨Lay out the S tube display, adjusting *p_height* appropriately 30a⟩≡ (27b)

$$
\begin{array}{l}
\{\\
\quad \mathit{p\_height} \overset{+}{\leftarrow} \mathit{margin};\\
\quad \textbf{Dimension}\ d \leftarrow \mathit{parts}[\mathit{S\_TUBE}].\mathit{getPreferredSize}(\,);\\
\quad \textbf{int}\ \mathit{x\_pos} \leftarrow (\mathit{p\_width} - d.\mathit{width} + 1) \div 2;\\
\quad \textbf{int}\ \mathit{y\_pos} \leftarrow \mathit{p\_height};\\
\quad \mathit{p\_height} \overset{+}{\leftarrow} d.\mathit{height};\\
\quad \mathit{parts}[\mathit{S\_TUBE}].\mathit{setBounds}(\mathit{x\_pos},\ \mathit{y\_pos},\ d.\mathit{width},\ d.\mathit{height});\\
\quad d \leftarrow \mathit{parts}[\mathit{S\_LABEL}].\mathit{getPreferredSize}(\,);\\
\quad \mathit{x\_pos} \overset{-}{\leftarrow} \mathit{parts}[\mathit{S\_BOX}].\mathit{getPreferredSize}(\,).\mathit{width};\\
\quad \mathit{parts}[\mathit{S\_LABEL}].\mathit{setBounds}(\mathit{x\_pos} - d.\mathit{width},\ \mathit{y\_pos},\ d.\mathit{width},\ d.\mathit{height});\\
\quad \mathit{y\_pos} \overset{+}{\leftarrow} (d.\mathit{height} + 1) \div 2;\\
\quad d \leftarrow \mathit{parts}[\mathit{S\_BOX}].\mathit{getPreferredSize}(\,);\\
\quad \mathit{parts}[\mathit{S\_BOX}].\mathit{setBounds}(\mathit{x\_pos},\ \mathit{y\_pos} - (d.\mathit{height} + 1) \div 2,\ d.\mathit{width},\ d.\mathit{height});\\
\quad \mathit{p\_height} \overset{+}{\leftarrow} \mathit{margin};\\
\}
\end{array}
$$

Defines:
x_pos, used in chunk 29.

Uses **Dimension**, *getPreferredSize* 41c, *height*, *margin* 27b, *parts* 24, *p_height* 27b, *p_width* 27b, *S_BOX* 24, *setBounds*, *S_LABEL* 24, *S_TUBE* 24, and *width*.

Display Tubes We use display tubes to show the contents of the Williams tubes.

30b ⟨* 1⟩+≡ ◁27a 34a▷

class Display_Tube
...**extends Canvas** {
⟨**Display_Tube**'s members and methods 31a⟩;
}

Defines:
Display_Tube, used in chunks 23b and 25–27.

Uses **Canvas**.

Each display tube has a fixed size, with a black background. Each displayed line contains 32 bits, so the width is constant, while the height depends on the number of lines in the tube. We use the kludgey "⟨**Display_Tube**'s defined constants⟩" chunk because the compiler insists that *BIT_WIDTH* be declared before it is used in the definition of *Width*.

Each display tube is connected to the actual Williams Tube that it displays. The display is initially enabled, so its contents will be visible.

31a ⟨**Display_Tube**'s members and methods 31a⟩≡ (30b) 32a▷

```
⟨Make this a fixed-size (Width × Height) component 41c⟩;
⟨Display_Tube's defined constants 31b⟩;
private final static int Width ← BIT_WIDTH × Line.BITS_PER_LINE;
private int Height;
private Williams_Tube actual_tube;
private int num_lines;
public Display_Tube(Williams_Tube t) {
  setBackground(Color.black);
  actual_tube ← t;
  num_lines ← t.get_Num_Lines( );
  Height ← num_lines × LINE_HEIGHT;
  setEnabled(true);
}
```

Defines:
actual_tube, used in chunk 32a.
Display_Tube, used in chunks 23b and 25–27.
Height, used in chunk 41c.
num_lines, used in chunk 32a.
Width, used in chunks 40a and 41c.

Uses *BITS_PER_LINE* 4a, *BIT_WIDTH* 31b, *black*, **Color**, *get_Num_Lines* 5a, **Line** 4a, *LINE_HEIGHT* 31b, *setBackground*, *setEnabled* 33b, and **Williams_Tube** 3b.

We allow 8×8 pixels for each bit in a line. "On" bits will be represented by a dash, "off" bits by a dot. Dashes will be 5 pixels wide, dots 2. Both will be 2 pixels high.

31b ⟨**Display_Tube**'s defined constants 31b⟩≡ (31a)

```
private final static int LINE_HEIGHT ← 8;
private final static int BIT_WIDTH ← 8;
private final static int BIT_HEIGHT ← 2;
private final static int DASH_WIDTH ← 5;
private final static int DOT_WIDTH ← 2;
```

Defines:
BIT_HEIGHT, used in chunk 33a.
BIT_WIDTH, used in chunks 31–33.
DASH_WIDTH, used in chunk 33a.
DOT_WIDTH, used in chunk 33a.
LINE_HEIGHT, used in chunks 31–33.

We paint a display tube simply by drawing a "blob" for each bit on each line, but only when the tube is "enabled." We need to investigate the possibility of only redrawing the area that's actually changed (the **Graphics**.*getClipRect* method may be helpful here).

32a ⟨**Display_Tube**'s members and methods 31a⟩+≡ (30b) ◁31a 32b▷

```
public void paint(Graphics g) {
  if (is_enabled) {
    for (int i ← 0; i < num_lines; i++) {
      int n ← actual_tube.get_Line_Value(i);
      for (int j ← 0, mask ← #1; j < Line.BITS_PER_LINE; j++,
            mask ≪⇐ 1) {
        blob(g, i, j, (n & mask) ≠ 0);
      }
    }
  } else {
    g.setColor(Color.black);
    g.fillRect(0, 0, Line.BITS_PER_LINE × BIT_WIDTH,
        num_lines × LINE_HEIGHT);
  }
}
```

Defines:
paint, used in chunk 32b.

Uses *actual_tube* 31a, *BITS_PER_LINE* 4a, *BIT_WIDTH* 31b, *black*, *blob* 33a, **Color**, *fillRect*, *get_Line_Value* 4c, **Graphics**, *is_enabled* 33b, **Line** 4a, *LINE_HEIGHT* 31b, *num_lines* 31a, and *setColor*.

Since blobbing takes care of redrawing black over where a dash used to be, there is no need to fill the display with the background color when it is updated.

32b ⟨**Display_Tube**'s members and methods 31a⟩+≡ (30b) ◁32a 32c▷

```
public void update(Graphics g) {
  paint(g);
}
```

Uses **Graphics** and *paint* 32a 40a 41a.

Most repainting of the display tubes involves a single line. It therefore makes sense to provide a *repaint* method that allows us to specify which line we want to repaint, translating the line number into the appropriate boundaries for the real *repaint* call.

Since this version of *repaint* is only invoked by the simulator, not the Java runtime system, we can avoid all repainting when the display is turned off, which should significantly speed up execution.

32c ⟨**Display_Tube**'s members and methods 31a⟩+≡ (30b) ◁32b 33a▷

```
public void repaint(int n) {
  if (is_enabled)
    repaint(0, n × LINE_HEIGHT, Line.BITS_PER_LINE × BIT_WIDTH,
        LINE_HEIGHT);
}
```

Defines:
repaint, used in chunks 23b, 27a, and 40b.

Uses *BITS_PER_LINE* 4a, *BIT_WIDTH* 31b, *is_enabled* 33b, **Line** 4a, and *LINE_HEIGHT* 31b.

For now we take the easy way out and use *fillRect* to draw each blob as needed. This should probably be replaced by bitmaps (for efficiency).

33a ⟨**Display_Tube**'s members and methods 31a⟩+≡ (30b) ◁32c 33b▷

```
private static void blob(Graphics g, int row, int col, boolean val) {
  g.setColor(Color.white);
  g.fillRect(2 + col × BIT_WIDTH, 4 + row × LINE_HEIGHT, DOT_WIDTH,
      BIT_HEIGHT);
  if (¬val)
    g.setColor(Color.black);
  g.fillRect(2 + col × BIT_WIDTH + DOT_WIDTH, 4 + row × LINE_HEIGHT,
      DASH_WIDTH − DOT_WIDTH, BIT_HEIGHT);
}
```

Defines:
blob, used in chunk 32a.

Uses *BIT_HEIGHT* 31b, *BIT_WIDTH* 31b, *black*, **Color**, *DASH_WIDTH* 31b, *DOT_WIDTH* 31b, *fillRect*, **Graphics**, *LINE_HEIGHT* 31b, *setColor*, and *white*.

Naturally, we need a way to turn a tube display on or off.

33b ⟨**Display_Tube**'s members and methods 31a⟩+≡ (30b) ◁33a

```
private boolean is_enabled;
public void setEnabled(boolean b) {
  is_enabled ← b;
}
```

Defines:
is_enabled, used in chunk 32.
setEnabled, used in chunks 27a, 31a, 34b, and 36b.

5.2.2 The Typewriter Panel

The typewriter panel (Figure 3) contains 40 "press-buttons" used to enter information into the store. Only the first 32 are enabled, since each line in the store contains only 32 bits.

0	5	10	15	20	25	30	~~35~~
1	6	11	16	21	26	31	~~36~~
2	7	12	17	22	27	~~32~~	~~37~~
3	8	13	18	23	28	~~33~~	~~38~~
4	9	14	19	24	29	~~34~~	~~39~~

Figure 3: The Typewriter Panel

To prevent the buttons from expanding to fill the available space, we create a separate panel, p, which will actually contain the buttons, and will be centered within the typewriter panel.

34a ⟨* 1⟩+≡ ◁30b 35a▷
```
class Typewriter_Panel
...extends Panel {
  public Typewriter_Panel( ) {
    Panel p ← new Panel( );
    add(p);
    ⟨Create a grid of 40 buttons in p 34b⟩;
  }
}
```
Defines:
  Typewriter_Panel, used in chunk 22.
Uses *add* and **Panel**.

Buttons 32–39 are disabled.

34b ⟨Create a grid of 40 buttons in p 34b⟩≡ (34a)
```
p.setLayout(new GridLayout(5, 8));
for (int i ← 0; i < 5; i++) {
  for (int j ← 0; j < 8; j++) {
    int n ← i + (j × 5);
    Button b ← new CB_Button("" + n,
        new Typewriter_Callback(n));
    p.add(b);
    if (n ≥ Line.BITS_PER_LINE)
      b.setEnabled(false);
  }
}
```
Uses *add*, *BITS_PER_LINE* 4a, **Button**, **CB_Button** 19d 20b, **GridLayout**, **Line** 4a, *setEnabled* 33b, *setLayout*, and **Typewriter_Callback** 18b.

5.2.3 The Staticisor Panel

The 16 staticisor switches[4] and the automatic/manual operation switch are included on the staticisor panel. Figure 4 shows how this panel is laid out.

S_0–S_4	S_5–S_{12}	S_{13}–S_{15}	*Auto/ Manual switch*

Figure 4: The Staticisor Panel

35a ⟨* 1⟩+≡ ◃34a 37a▹

```
class Stat_Panel
...extends Panel {
   ⟨Stat_Panel's members and methods 35b⟩;
}
```

Defines:
Stat_Panel, used in chunk 22.

Uses **Panel**.

35b ⟨**Stat_Panel**'s members and methods 35b⟩≡ (35a) 36b▹

```
public Stat_Panel( ) {
   ⟨Add switches S0–S15 36a⟩;
   ⟨Add the Auto/Manual switch 35c⟩;
}
```

Defines:
Stat_Panel, used in chunk 22.

The automatic/manual operation switch is a simple checkbox group, initially in the "manual" position.

35c ⟨Add the Auto/Manual switch 35c⟩≡ (35b)

```
{
   Panel p ← new Panel( );
   p.setLayout(new GridLayout(2, 1));
   CheckboxGroup g ← new CheckboxGroup( );
   p.add(new CB_Checkbox("Automatic", g, false,
        new Manual_Mode_Callback(false)));
   p.add(new CB_Checkbox("Manual", g, true,
        new Manual_Mode_Callback(true)));
   ssem.machine.set_Manual(true);
   add(p);
}
```

Uses *add*, **CB_Checkbox** 20c 20e 21a, **CheckboxGroup**, **GridLayout**, *machine* 2a, **Manual_Mode_Callback** 13a, **Panel**, *setLayout*, *set_Manual* 12e, and **ssem** 1 2a.

[4] Actually, only 8 of these switches (S_0–S_4 and S_{13}–S_{15}) are enabled. The others are included only for completeness.

We use a "helper" function, *stat_switches*, to create panels for each of the 3 groups of staticisor switches. Switches 0–4 and 13–15 are enabled; switches 5–12 are disabled.

36a ⟨Add switches S_0–S_{15} 36a⟩≡ (35b)

```
{
  Panel p ← new Panel( );
  p.add(stat_switches(0, 4, true));
  p.add(stat_switches(5, 12, false));
  p.add(stat_switches(13, 15, true));
  add(p);
}
```

Uses *add*, **Panel**, and *stat_switches* 36b.

The *stat_switches* function simply creates the requisite number of switches, disabling the switches, if necessary. We outline the panel because the Win95 JDK does not "gray out" disabled checkboxes, and there is no other way to separate the unused switches from the used ones.

36b ⟨**Stat_Panel**'s members and methods 35b⟩+≡ (35a) ◁35b

```
static Outlined_Panel stat_switches(int first, int last, boolean working) {
  Stat_Switch s;
  Outlined_Panel p ← new Outlined_Panel( );
  for (int i ← first; i ≤ last; i++) {
    p.add(s ← new Stat_Switch(i));
    if (¬working)
      s.setEnabled(false);
  }
  return p;
}
```

Defines:
stat_switches, used in chunk 36a.

Uses *add*, **Outlined_Panel** 40e, *setEnabled* 33b, and **Stat_Switch** 37a.

A staticisor switch is a simple checkbox group labeled with an 'S' followed by an integer, and initially set in the "closed" (i.e., "on") position.

37a ⟨* 1⟩+≡ ◁35a 37b▷

```
class Stat_Switch
...extends Panel {
  public Stat_Switch(int switch_num) {
    setLayout(new GridLayout(3, 1));
    Label l ← new Label("S" + switch_num);
    l.setAlignment(Label.LEFT);
    add(l);
    CheckboxGroup g ← new CheckboxGroup( );
    add(new CB_Checkbox("", g, true,
        new Stat_Switch_Callback(switch_num, true)));
    add(new CB_Checkbox("", g, false,
        new Stat_Switch_Callback(switch_num, false)));
    ssem.machine.set_Stat_Switch_Bit(switch_num);
  }
}
```

Defines:
Stat_Switch, used in chunk 36b.

Uses *add*, **CB_Checkbox** 20c 20e 21a, **CheckboxGroup**, **GridLayout**, **Label**, *LEFT*, *machine* 2a, **Panel**, *setAlignment*, *setLayout*, *set_Stat_Switch_Bit* 14a, **ssem** 1 2a, and **Stat_Switch_Callback** 14b.

5.2.4 The Control Panel

The control panel contains the pre-pulse and write/erase switches as well as the KSP, KLC, KCC, KAC, KEC, and KSC keys, and the stop lamp. Since the format is not important, we use the default "flow" layout. A grid bag layout would probably make for a better-looking interface.

37b ⟨* 1⟩+≡ ◁37a 39f▷

```
class Control_Panel
...extends Panel {
  public Control_Panel( ) {
    ⟨Add the pre-pulse switch 38a⟩;
    ⟨Add the KSP key 38c⟩;
    ⟨Add the KLC key 38d⟩;
    ⟨Add the KSC key 39a⟩;
    ⟨Add the KAC key 39b⟩;
    ⟨Add the KCC key 38e⟩;
    ⟨Add the KEC key 39c⟩;
    ⟨Add the write/erase switch 38b⟩;
    ⟨Add the stop lamp 39d⟩;
  }
  ⟨Control_Panel's members and methods 39e⟩;
}
```

Defines:
Control_Panel, used in chunk 22.

Uses **Panel**.

The pre-pulse and write/erase switches are labeled checkbox groups. The former is initially off, the latter in the "write" position. The pre-pulse key was labeled 'CS', for "completion signal," a term "from the early days when the designers thought of the pulse which initiates a new instruction as actually completing the previous instruction" [2].

38a ⟨Add the pre-pulse switch 38a⟩≡ (37b)

```
{
  Panel p ← new Panel( );
  p.setLayout(new GridLayout(3, 1));
  p.add(new Label("␣CS"));
  CheckboxGroup g ← new CheckboxGroup( );
  p.add(new CB_Checkbox("Run", g, false,
      new Start_Exec_Callback( )));
  p.add(new CB_Checkbox("Stop", g, true,
      new Stop_Exec_Callback( )));
  ssem.machine.set_Prepulse(false);
  add(p);
}
```

Uses *add*, **CB_Checkbox** 20c 20e 21a, **CheckboxGroup**, **GridLayout**, **Label**, *machine* 2a, **Panel**, *setLayout*, *set_Prepulse* 10d, **ssem** 1 2a, **Start_Exec_Callback** 15a, and **Stop_Exec_Callback** 15b.

38b ⟨Add the write/erase switch 38b⟩≡ (37b)

```
{
  Panel p ← new Panel( );
  p.setLayout(new GridLayout(3, 1));
  p.add(new Label("Write/Erase"));
  CheckboxGroup g ← new CheckboxGroup( );
  p.add(new CB_Checkbox("Write", g, true,
      new Write_Flag_Callback(true)));
  p.add(new CB_Checkbox("Erase", g, false,
      new Write_Flag_Callback(false)));
  ssem.machine.set_Write_Flag(true);
  add(p);
}
```

Uses *add*, **CB_Checkbox** 20c 20e 21a, **CheckboxGroup**, **GridLayout**, **Label**, *machine* 2a, **Panel**, *setLayout*, *set_Write_Flag* 17b, **ssem** 1 2a, and **Write_Flag_Callback** 18a.

The keys are simple callback buttons. The KSP button was labeled 'KC', for "key completion" [2, 6].

38c ⟨Add the KSP key 38c⟩≡ (37b)

```
add(new CB_Button("KC", new KSP_Callback( )));
```

Uses *add*, **CB_Button** 19d 20b, and **KSP_Callback** 13b.

38d ⟨Add the KLC key 38d⟩≡ (37b)

```
add(new CB_Button("KLC", new KLC_Callback( )));
```

Uses *add*, **CB_Button** 19d 20b, and **KLC_Callback** 16b.

38e ⟨Add the KCC key 38e⟩≡ (37b)

```
add(new CB_Button("KCC", new KCC_Callback( )));
```

Uses *add*, **CB_Button** 19d 20b, and **KCC_Callback** 16a.

39a ⟨Add the KSC key 39a⟩≡ (37b)

add(**new CB_Button**("KSC", **new KSC_Callback**( )));

Uses *add*, **CB_Button** 19d 20b, and **KSC_Callback** 15c.

39b ⟨Add the KAC key 39b⟩≡ (37b)

add(**new CB_Button**("KAC", **new KAC_Callback**( )));

Uses *add*, **CB_Button** 19d 20b, and **KAC_Callback** 17a.

39c ⟨Add the KEC key 39c⟩≡ (37b)

add(**new CB_Button**("KEC", **new KEC_Callback**( )));

Uses *add*, **CB_Button** 19d 20b, and **KEC_Callback** 17a.

The stop lamp is full-fledged object, since it will need to paint itself differently, depending on its state.

39d ⟨Add the stop lamp 39d⟩≡ (37b)

add(*lamp* ← **new Stop_Lamp**( ));

Uses *add*, *lamp* 23a 39e, and **Stop_Lamp** 39e 39f.

39e ⟨**Control_Panel**'s members and methods 39e⟩≡ (37b)

```
private Stop_Lamp lamp;
public Stop_Lamp get_Lamp( ) {
  return lamp;
}
```

Defines:
lamp, used in chunks 22 and 39d.
Stop_Lamp, used in chunks 23a and 39d.

Uses *get_Lamp* 23a.

The Stop Lamp The stop lamp is a canvas with a fixed square size (currently 30×30).

39f ⟨* 1⟩+≡ ◁37b 40e▷

```
class Stop_Lamp
... extends Canvas {
  ⟨Stop_Lamp's members and methods 40a⟩;
  private final static int Width ← 30;
  private final static int Height ← 30;
  ⟨Make this a fixed-size (Width × Height) component 41c⟩;
}
```

Defines:
Height, used in chunk 41c.
Stop_Lamp, used in chunks 23a and 39d.
Width, used in chunks 40a and 41c.

Uses **Canvas**.

The lamp can either be on or off. When it is off, a black circle is displayed; when on, a red one.

40a ⟨**Stop_Lamp**'s members and methods 40a⟩≡ (39f) 40b▷

```
private boolean lamp_is_on;
public void paint(Graphics g) {
  if (lamp_is_on)
    g.setColor(Color.red);
  else
    g.setColor(Color.black);
  g.fillOval(0, 0, Width, Width);
}
```

Defines:
lamp_is_on, used in chunk 40b.
paint, used in chunk 32b.

Uses *black*, **Color**, *fillOval*, **Graphics**, *red*, *setColor*, and *Width* 31a 39f.

We make it possible for the simulator to illuminate and turn off the stop lamp, as necessary.

40b ⟨**Stop_Lamp**'s members and methods 40a⟩+≡ (39f) ◁40a

```
public void illuminate(boolean b) {
  lamp_is_on ← b;
  repaint( );
}
```

Defines:
illuminate, used in chunk 40.

Uses *lamp_is_on* 40a and *repaint* 32c.

40c ⟨Illuminate the stop lamp 40c⟩≡ (8d)

```
ssem.iface.get_Lamp( ).illuminate(true);
```

Uses *get_Lamp* 23a, *iface* 2a, *illuminate* 40b, and **ssem** 1 2a.

40d ⟨Turn off the stop lamp 40d⟩≡ (13b 15a)

```
ssem.iface.get_Lamp( ).illuminate(false);
```

Uses *get_Lamp* 23a, *iface* 2a, *illuminate* 40b, and **ssem** 1 2a.

5.2.5 Outlined Panels

For clarity, some of the panels in the display are outlined. An outlined panel is simply a regular panel with 2 methods redefined.

40e ⟨* 1⟩+≡ ◁39f

```
class Outlined_Panel
...extends Panel {
  ⟨Outlined_Panel's redefined Panel methods 41a⟩;
}
```

Defines:
Outlined_Panel, used in chunk 36b.

Uses **Panel**.

The *paint* method draws a single-pixel black outline around the panel, after painting it normally.

41a ⟨**Outlined_Panel**'s redefined **Panel** methods 41a⟩≡ (40e) 41b▹

```
public void paint(Graphics g) {
  super.paint(g);
  Dimension d ← getSize( );
  g.setColor(Color.black);
  g.drawRect(0, 0, d.width − 1, d.height − 1);
}
```

Defines:
paint, used in chunk 32b.

Uses *black*, **Color**, **Dimension**, *drawRect*, *getSize* 41c, **Graphics**, *height*, *setColor*, and *width*.

The *getInsets* method allows space for the panel's outline.

41b ⟨**Outlined_Panel**'s redefined **Panel** methods 41a⟩+≡ (40e) ◃41a

```
public Insets getInsets( ) {
  return new Insets(1, 1, 1, 1);
}
```

Uses **Insets**.

5.2.6 Fixed Size Components

We can fix the size of a component simply by redefining the object's *getMinimumSize*, *getPreferredSize*, and *getSize* methods to return a fixed value. Each such component must define the members *Width* and *Height*.

41c ⟨Make this a fixed-size (*Width* × *Height*) component 41c⟩≡ (31a 39f)

```
public Dimension getMinimumSize( ) {
  return new Dimension(Width, Height);
}
public Dimension getPreferredSize( ) {
  return getMinimumSize( );
}
public Dimension getSize( ) {
  return getMinimumSize( );
}
```

Defines:
getPreferredSize, used in chunks 28–30.
getSize, used in chunk 41a.

Uses **Dimension**, *Height* 31a 39f, and *Width* 31a 39f.

5.2.7 Grid Bags

The "grid bag" layout manager allows the most flexibility, but it also requires a lot of boilerplate code to get working. Here we provide that boilerplate.

We need both a grid bag layout manager, and a set of constraints. We'll use the variables *gbl* and *gbc*, respectively. These will, of course, be local to whichever block they are used in. Since we are going to pack everything before we show it and disallow resizing, we don't want to allow any filling. If we should need anything special, we can change the constraints after setting up the grid bag.

42a ⟨Set up a grid bag layout, *gbl*, with constraints *gbc* 42a⟩≡ (22)

```
GridBagLayout gbl ← new GridBagLayout( );
setLayout(gbl);
GridBagConstraints gbc ← new GridBagConstraints( );
gbc.weightx ← gbc.weighty ← 0;
gbc.fill ← GridBagConstraints.NONE;
```

Defines:
gbc, used in chunks 22 and 42b.
gbl, used in chunk 42b.

Uses *fill*, **GridBagConstraints**, **GridBagLayout**, *NONE*, *setLayout*, *weightx*, and *weighty*.

42b ⟨Add component *x* to grid bag *gbl*, with constraints *gbc* 42b⟩≡ (22)

```
gbl.setConstraints(x, gbc);
add(x);
```

Uses *add*, *gbc* 42a, *gbl* 42a, and *setConstraints*.

6 Known Flaws in the Simulator

It is rarely possible, or even desirable, for a simulator to match the behavior of the original exactly in all situations. Although we have tried to make this simulator as accurate a model as possible, there are a number of instances in which we have deliberately introduced inaccuracies in the interface or operation of the machine. None of these affect the operation of the simulator under "normal" circumstances; they all involve "exceptional" situations or modern conventions at odds with the original interface.

- We provide separate displays for each of the 3 tubes. There was only one display tube in the SSEM, which could be used to display the contents of one tube at a time.
- Non-zero bits in the function part (bits 13–15) of the C.I. are ignored. In the original, the machine would "hang."
- The simulator does not brighten the "action line" (the line specified for an operation by the staticisor switches or address portion of an instruction) as the original did.
- The simulator does not require a manual pre-pulse before restarting after a STOP instruction. Simply resetting the Pre-pulse switch to "Run" will cause execution to resume.
- The position of the Write/Erase switch is ignored during program execution. In the original, this would corrupt the program in the store as each line became activated.
- The switches in the simulator interface use the up position to mean on, down for off. The original SSEM used the opposite convention.
- The KLC key and the typewriter buttons affect only the line specified by the staticisor switches, whether the simulator is running a program or not. When the original machine was running, all lines accessed at the time would be affected.
- We provide the KAC and KEC keys, even though these were not physically connected in June 1948. The original also provided the KBC and KMC keys, which were never connected.

Acknowledgements

Special thanks are due to Chris Burton and Bill Purvis of the SSEM Rebuild Project, whose feedback on the details of the machine and on the progress of the rebuild were invaluable in making the simulator as accurate a model as it is. Any mistakes, of course, are my own.

Thanks are also due to Martin Campbell-Kelly for his EDSAC simulators. Much of the "look and feel" of this simulator is based on his work.

References

[1] Chris P. Burton. Personal communication.

[2] Chris P. Burton. *The Manchester University Small-Scale Experimental Machine Programmer's Reference Manual.* Computer Conservation Society, August 1997. `http://www.cs.man.ac.uk/prog98/ssemref.html`.

[3] Martin Campbell-Kelly. Programming the Mark I: Early programming activity at the University of Manchester. *Annals of the History of Computing*, 2(2):130–168, April 1980.

[4] S. H. Lavington. *A History of Manchester Computers.* NCC Publications, Manchester, UK, 1975.

[5] S. H. Lavington. *Early British Computers.* Digital Press, Bedford MA, USA, 1980.

[6] W. H. Purvis. Personal communication.

[7] Brian Randell, editor. *The Origins of Digital Computers,* Selected Papers. Springer-Verlag, 1973.

[8] F. C. Williams and T. Kilburn. Electronic digital computers. *Nature*, 162:487, September 1948. Reprinted in Randell [7, pages 387–388].

[9] F. C. Williams, T. Kilburn, and G. C. Tootill. Universal high-speed digital computers: A small-scale experimental machine. *Proceedings of the IEE*, 98, Part II(61):13–28, February 1951. Also available as `ftp://ftp.cs.man.ac.uk/pub/CCS-Archive/misc/IEE1.doc`.

Chunk Index

⟨* 1⟩ 1, 3b, 4a, 13a, 13b, 14b, 15a, 15b, 15c, 16a, 16b, 17a, 18a, 18b, 19b, 19c, 19d, 20c, 26, 27a, 30b, 34a, 35a, 37a, 37b, 39f, 40e
⟨Add component x to grid bag *gbl*, with constraints *gbc* 42b⟩ 22, 42b
⟨Add switches *S*_0–*S*_15 36a⟩ 35b, 36a
⟨Add the Auto/Manual switch 35c⟩ 35b, 35c
⟨Add the KAC key 39b⟩ 37b, 39b
⟨Add the KCC key 38e⟩ 37b, 38e
⟨Add the KEC key 39c⟩ 37b, 39c
⟨Add the KLC key 38d⟩ 37b, 38d
⟨Add the KSC key 39a⟩ 37b, 39a
⟨Add the KSP key 38c⟩ 37b, 38c
⟨Add the pre-pulse switch 38a⟩ 37b, 38a
⟨Add the stop lamp 39d⟩ 37b, 39d
⟨Add the write/erase switch 38b⟩ 37b, 38b
⟨Adjust the layout of the current display panel 27b⟩ 26, 27b
⟨Allow the window to be closed 21c⟩ 21b, 21c
⟨Build the interface 22⟩ 21b, 22
⟨Compute *p_width* and *p_height* for the top part of the panel 28a⟩ 27b, 28a
⟨Create a grid of 40 buttons in *p* 34b⟩ 34a, 34b
⟨Execute a single instruction 9a⟩ 9a, 10c
⟨factorization program 11⟩ 11, 12d
⟨Illuminate the stop lamp 40c⟩ 8d, 40c
⟨Increment C.I. 9b⟩ 9a, 9b
⟨Initialize the SSEM 12c⟩ 2a, 12c
⟨**CB_Button**'s members and methods 20a⟩ 19d, 20a, 20b
⟨**CB_Checkbox**'s members and methods 20d⟩ 20c, 20d, 20e, 21a
⟨**Control_Panel**'s members and methods 39e⟩ 37b, 39e
⟨**Display_Panel**'s members and methods 25⟩ 25, 26
⟨**Display_Tube**'s defined constants 31b⟩ 31a, 31b
⟨**Display_Tube**'s members and methods 31a⟩ 30b, 31a, 32a, 32b, 32c, 33a, 33b
⟨**Line**'s members and methods 4b⟩ 4a, 4b
⟨**Outlined_Panel**'s redefined **Panel** methods 41a⟩ 40e, 41a, 41b
⟨**ssem_Interface**'s members and methods 21b⟩ 19b, 21b, 21d, 23a, 23b
⟨**ssem**'s members and methods 2a⟩ 1, 2a, 3a, 5b, 6a, 8c, 10a, 10c, 10d, 12b, 12e, 14a, 17b
⟨**Stat_Panel**'s members and methods 35b⟩ 35a, 35b, 36b
⟨**Stop_Lamp**'s members and methods 40a⟩ 39f, 40a, 40b
⟨**Williams_Tube**'s members and methods 4c⟩ 3b, 4c, 5a
⟨Keep a list, *parts*, of display panel components 24⟩ 24, 26
⟨Lay out the A and C tube displays 28b⟩ 27b, 28b
⟨Lay out the A tube display, adjusting *xpos* 29a⟩ 28b, 29a
⟨Lay out the C tube display, adjusting *xpos* 29b⟩ 28b, 29b
⟨Lay out the S tube display, adjusting *p_height* appropriately 30a⟩ 27b, 30a
⟨Load the factorization program 12d⟩ 12c, 12d
⟨Make this a fixed-size ($Width \times Height$) component 41c⟩ 31a, 39f, 41c
⟨Move the contents of P.I. to the staticisor 9d⟩ 9a, 9d
⟨Move the instruction specified by C.I. to P.I. 9c⟩ 9a, 9c

⟨Obey the instruction in the staticisor 10b⟩ 9a, 10b
⟨Obey the *op*/*addr* instruction 6b⟩ 6b, 7a, 7b, 7c, 8a, 8b, 8d, 10b
⟨Package imports 19a⟩ 1, 19a, 21e
⟨Parse argument *args*[*i*] 12a⟩ 2b, 12a
⟨Parse command line arguments, *args*[] 2b⟩ 2a, 2b
⟨Set up a grid bag layout, *gbl*, with constraints *gbc* 42a⟩ 22, 42a
⟨Turn off the stop lamp 40d⟩ 13b, 15a, 40d

Identifier Index

A_BOX: 24, 26, 29a
ActionEvent: 20a
ActionListener: 19d
actionPerformed: 20a
actual_tube: 31a, 32a
add: 26, 34a, 34b, 35c, 36a, 36b, 37a, 38a, 38b, 38c, 38d, 38e, 39a, 39b, 39c, 39d, 42b
addActionListener: 20b
addItemListener: 20e, 21a
addr: 6b, 7a, 7b, 7c, 8a, 10b
ADDR_BITS: 6a, 10b
ADDR_MASK: 6a, 9c, 10b, 16b, 18b
addWindowListener: 21c
a_display: 22, 23b
adjust: 22, 26
A_LABEL: 24, 26, 28a, 29a
ALL_ONES: 4a, 9d, 14a
args: 2a, 2b, 12a
a_tube: 3a, 5b, 7b, 7c, 8a, 8b
A_TUBE: 24, 25, 26, 29a
awt: 19a, 21e
AWTEvent: 13a, 13b, 14b, 15a, 15b, 15c, 16a, 16b, 17a, 18a, 18b, 19c, 27a
bit: 14b
BIT_HEIGHT: 31b, 33a
BITS_PER_LINE: 4a, 31a, 32a, 32c, 34b
BIT_WIDTH: 31a, 31b, 32a, 32c, 33a
black: 31a, 32a, 33a, 40a, 41a
blob: 32a, 33a
Button: 19d, 34b
Callback: 13a, 13b, 14b, 15a, 15b, 15c, 16a, 16b, 17a, 18a, 18b, 19c, 19d, 20b, 20c, 20e, 21a, 27a
Canvas: 30b, 39f
cb: 19d, 20a, 20b, 20c, 20d, 20e, 21a
CB_Button: 19d, 20b, 34b, 38c, 38d, 38e, 39a, 39b, 39c
CB_Checkbox: 20c, 20e, 21a, 26, 35c, 37a, 38a, 38b
C_BOX: 24, 26, 29b
c_display: 22, 23b
Checkbox: 20c, 27a
CheckboxGroup: 21a, 35c, 37a, 38a, 38b
CI: 3a, 6b, 7a, 9b, 9c
C_LABEL: 24, 26, 29b
clear_Tube: 5a, 15c, 16a, 17a
col: 33a
Color: 31a, 32a, 33a, 40a, 41a
Component: 22, 24, 26, 27a
Control_Panel: 22, 37b
C_TUBE: 24, 25, 26, 28a, 29b
c_tube: 3a, 5b, 6b, 7a, 9b, 9c, 9d
DASH_WIDTH: 31b, 33a
Dimension: 28a, 29a, 29b, 30a, 41a, 41c
Display_Panel: 22, 26
Display_Tube: 23b, 25, 26, 27a, 30b, 31a
DOT_WIDTH: 31b, 33a
drawRect: 41a
equals: 12a
event: 21e
exit: 21d
extra_gap: 27b, 28a, 28b
factor_pgm: 12d
fill: 42a
fillOval: 40a
fillRect: 32a, 33a
first: 36b
Frame: 19b
func: 13a, 13b, 14b, 15a, 15b, 15c, 16a, 16b, 17a, 18a, 18b, 19c, 20a, 20d, 27a
FUNC_BITS: 6a

FUNC_MASK: 6a, 10b
gbc: 22, 42a, 42b
gbl: 42a, 42b
get_A_Display: 22, 25
get_A_Tube: 5b, 16a, 17a, 26
get_C_Display: 22, 25
get_C_Tube: 5b, 16a, 17a, 26
getInsets: 41b
get_Lamp: 22, 23a, 39e, 40c, 40d
get_Line_Value: 4c, 6b, 7a, 7b, 7c, 8a, 8b, 9b, 9c, 9d, 18b, 32a
get_Manual: 9d, 12e
getMinimumSize: 41c
get_Num_Lines: 5a, 31a
getPreferredSize: 28a, 29a, 29b, 30a, 41c
get_S_Display: 22, 25
getSize: 41a, 41c
getSource: 27a
getState: 27a
get_Stat_Switches: 9d, 14a, 16b, 18b
get_S_Tube: 5b, 15c, 16b, 17a, 18b, 26
get_Value: 4b, 4c
get_Write_Flag: 17b, 18b
Graphics: 32a, 32b, 33a, 40a, 41a
GridBagConstraints: 42a
GridBagLayout: 42a
gridheight: 22
GridLayout: 34b, 35c, 37a, 38a, 38b
gridwidth: 22
gridx: 22
gridy: 22
Height: 31a, 39f, 41c
height: 28a, 29a, 29b, 30a, 41a
iface: 2a, 6b, 7a, 7b, 7c, 8a, 9b, 9c, 15c, 16a, 16b, 17a, 18b, 40c, 40d
illuminate: 40b, 40c, 40d
Insets: 41b
is_enabled: 32a, 32c, 33b
ItemEvent: 20d
ItemListener: 20c
itemStateChanged: 20d
java: 19a, 21e
KAC_Callback: 17a, 39b
KCC_Callback: 16a, 38e
KEC_Callback: 17a, 39c
KLC_Callback: 16b, 38d
KSC_Callback: 15c, 39a
KSP_Callback: 13b, 38c
Label: 26, 37a, 38a, 38b
lamp: 22, 23a, 39d, 39e
lamp_is_on: 40a, 40b
last: 36b
LEFT: 37a
length: 2b, 5a, 12d
Line: 3b, 4a, 9d, 14a, 31a, 32a, 32c, 34b
LINE_HEIGHT: 31a, 31b, 32a, 32c, 33a
lines: 3b, 4c, 5a
load_demo: 12a, 12b, 12c
machine: 2a, 13a, 13b, 14b, 15a, 15b, 15c, 16a, 16b, 17a, 18a, 18b, 26, 35c, 37a, 38a, 38b
main: 2a
manual_mode: 12e
Manual_Mode_Callback: 13a, 35c
margin: 27b, 28a, 28b, 30a
mask: 18b, 32a
Math: 28a
max: 28a
NONE: 42a
NORM_PRIORITY: 2a
num_lines: 31a, 32a
NUM_PARTS: 24, 26
op: 10b
Outlined_Panel: 36b, 40e
pack: 21b, 22
paint: 32a, 32b, 40a, 41a
Panel: 26, 34a, 35a, 35c, 36a, 37a, 37b, 38a, 38b, 40e
parts: 24, 25, 26, 28a, 29a, 29b, 30a
p_height: 27b, 28a, 29a, 29b, 30a
PI: 3a, 9c, 9d
pre_pulse: 10c, 10d
ProgramCloser: 21c, 21d
p_width: 27b, 28a, 30a
red: 40a
repaint: 23b, 27a, 32c, 40b
reset_Stat_Switch_Bit: 14a, 14b
resume: 13b, 15a
row: 33a
run: 10c
S_BOX: 24, 26, 30a
s_display: 22, 23b
setAlignment: 37a
setBackground: 31a
setBounds: 29a, 29b, 30a
setColor: 32a, 33a, 40a, 41a

setConstraints: 42b
setEnabled: 27a, 31a, 33b, 34b, 36b
setLayout: 26, 34b, 35c, 37a, 38a, 38b, 42a
set_Line_Value: 4c, 6b, 7a, 7b, 7c, 8a, 9b, 9c, 12d, 16b, 18b
set_Manual: 12e, 13a, 35c
set_Prepulse: 8d, 10d, 15a, 15b, 38a
setPriority: 2a
setResizable: 21b
setSize: 27b
set_Stat_Switch_Bit: 14a, 14b, 37a
setTitle: 21b
set_Value: 4b, 4c, 5a
setVisible: 21b
set_Write_Flag: 17b, 18a, 38b
S_LABEL: 24, 26, 30a
ssem: 1, 2a, 6b, 7a, 7b, 7c, 8a, 9b, 9c, 13a, 13b, 14b, 15a, 15b, 15c, 16a, 16b, 17a, 18a, 18b, 26, 35c, 37a, 38a, 38b, 40c, 40d
ssem_Interface: 2a, 19b, 21b
start: 2a
Start_Exec_Callback: 15a, 38a
staticisor: 9d, 10a, 10b
staticisor_switches: 14a
Stat_Panel: 22, 35a, 35b
Stat_Switch: 36b, 37a
Stat_Switch_Callback: 14b, 37a
stat_switches: 36a, 36b
Stop_Exec_Callback: 15b, 38a
Stop_Lamp: 23a, 39d, 39e, 39f
STORE_SIZE: 3a
String: 2a, 20b, 21a
S_TUBE: 24, 25, 26, 30a
s_tube: 3a, 5b, 6b, 7a, 7b, 7c, 8a, 9c, 12d
suspend: 10c
switch_num: 37a
System: 21d
test_flip_flop: 8b, 8c, 9b
Thread: 1, 2a
tube: 27a
Tube_Show_Callback: 26, 27a
Typewriter_Callback: 18b, 34b
Typewriter_Panel: 22, 34a
UNUSED_ADDR_BITS: 6a
update: 32b
update_A_Display: 7b, 8a, 16a, 17a, 23b
update_C_Display: 6b, 7a, 9b, 9c, 16a, 17a, 23b
update_S_Display: 7c, 15c, 16b, 17a, 18b, 23b
val: 13a, 14b, 18a, 33a
value: 4a, 4b
weightx: 42a
weighty: 42a
white: 33a
width: 28a, 29a, 29b, 30a, 41a
Width: 31a, 39f, 40a, 41c
Williams_Tube: 3a, 3b, 5b, 31a
WindowAdapter: 21d
windowClosing: 21d
WindowEvent: 21d
working: 36b
Write_Flag_Callback: 18a, 38b
write_mode: 17b
x_pos: 28b, 29a, 29b, 30a
yield: 10c
y_pos: 30a