1. Introduction. This program module contains brute-force implementations of the ten input/output primitives defined at the beginning of MMIX-SIM. The subroutines are grouped here as a separate package, because they are intended to be loaded with the pipeline simulator as well as with the simple simulator.

⟨ Preprocessor macros 2 ⟩
⟨ Type definitions 3 ⟩
⟨ External subroutines 4 ⟩
⟨ Global variables 6 ⟩
⟨ Subroutines 7 ⟩

2. Of course we include standard C library routines, and we set things up to accommodate older versions of C.

⟨ Preprocessor macros 2 ⟩ ≡
```
#include <stdio.h>
#include <stdlib.h>
#ifdef __STDC__
#define ARGS(list) list
#else
#define ARGS(list) ()
#endif
#ifndef FILENAME_MAX
#define FILENAME_MAX 256
#endif
#ifndef SEEK_SET
#define SEEK_SET 0
#endif
#ifndef SEEK_END
#define SEEK_END 2
#endif
```
This code is used in section 1.

3. The unsigned 32-bit type **tetra** must agree with its definition in the simulators.

⟨ Type definitions 3 ⟩ ≡
```
typedef unsigned int tetra;
typedef struct {
  tetra h, l;
} octa;      /* two tetrabytes make one octabyte */
```
See also section 5.

This code is used in section 1.

4. Three basic subroutines are used to get strings from the simulated memory and to put strings into that memory. These subroutines are defined appropriately in each simulator. We also use a few subroutines and constants defined in MMIX-ARITH.

```
⟨ External subroutines 4 ⟩ ≡
  extern char stdin_chr ARGS((void));
  extern int mmgetchars ARGS((char *buf, int size, octa addr, int stop));
  extern void mmputchars ARGS((unsigned char *buf, int size, octa addr));
  extern octa oplus ARGS((octa, octa));
  extern octa ominus ARGS((octa, octa));
  extern octa incr ARGS((octa, int));
  extern octa zero_octa;      /* zero_octa.h = zero_octa.l = 0 */
  extern octa neg_one;      /* neg_one.h = neg_one.l = -1 */
```

This code is used in section 1.

5. Each possible handle has a file pointer and a current mode.

```
⟨ Type definitions 3 ⟩ +≡
  typedef struct {
    FILE *fp;      /* file pointer */
    int mode;      /* [read OK] + 2[write OK] + 4[binary] + 8[readwrite] */
  } sim_file_info;
```

6. ⟨ Global variables 6 ⟩ ≡

```
  sim_file_info sfile[256];
```

See also sections 9 and 24.

This code is used in section 1.

7. The first three handles are initially open.

```
⟨ Subroutines 7 ⟩ ≡
  void mmix_io_init ARGS((void));
  void mmix_io_init()
  {
    sfile[0].fp = stdin, sfile[0].mode = 1;
    sfile[1].fp = stdout, sfile[1].mode = 2;
    sfile[2].fp = stderr, sfile[2].mode = 2;
  }
```

See also sections 8, 10, 11, 12, 14, 16, 18, 19, 20, 21, 22, and 23.

This code is used in section 1.

8. The only tricky thing about these routines is that we want to protect the standard input, output, and error streams from being preempted.

```
⟨ Subroutines 7 ⟩ +≡
  octa mmix_fopen ARGS((unsigned int, octa, octa));
  octa mmix_fopen(handle, name, mode)
      unsigned int handle;
      octa name, mode;
  {
    char name_buf[FILENAME_MAX];
    if (mode.h ∨ mode.l > 4) goto failure;
    if (mmgetchars(name_buf, FILENAME_MAX, name, 0) ≡ FILENAME_MAX) goto failure;
    if (sfile[handle].mode ≠ 0 ∧ handle > 2) fclose(sfile[handle].fp);
    sfile[handle].fp = fopen(name_buf, mode_string[mode.l]);
    if (¬sfile[handle].fp) goto failure;
    sfile[handle].mode = mode_code[mode.l];
    return zero_octa;      /* success */
  failure: sfile[handle].mode = 0;
    return neg_one;      /* failure */
  }
```

9. ⟨ Global variables 6 ⟩ +≡

```
  char *mode_string[] = {"r", "w", "rb", "wb", "w+b"};
  int mode_code[] = {#1, #2, #5, #6, #f};
```

10. If the simulator is being used interactively, we can avoid competition for *stdin* by substituting another file.

```
⟨ Subroutines 7 ⟩ +≡
  void mmix_fake_stdin ARGS((FILE *));
  void mmix_fake_stdin(f)
      FILE *f;
  {
    sfile[0].fp = f;      /* f should be open in mode "r" */
  }
```

11. ⟨ Subroutines 7 ⟩ +≡

```
  octa mmix_fclose ARGS((unsigned int));
  octa mmix_fclose(handle)
      unsigned int handle;
  {
    if (sfile[handle].mode ≡ 0) return neg_one;
    if (handle > 2 ∧ fclose(sfile[handle].fp) ≠ 0) return neg_one;
    sfile[handle].mode = 0;
    return zero_octa;      /* success */
  }
```

12. ⟨ Subroutines 7 ⟩ +≡

```
octa mmix_fread ARGS((unsigned int, octa, octa));
octa mmix_fread(handle, buffer, size)
    unsigned int handle;
    octa buffer, size;
{
  register unsigned char *buf;
  register unsigned int n;
  octa o;
  o = neg_one;
  if (¬(sfile[handle].mode & #1)) goto done;
  if (sfile[handle].mode & #8) sfile[handle].mode &= ∼#2;
  if (size.h) goto done;
  buf = (unsigned char *) calloc(size.l, sizeof(char));
  if (¬buf) goto done;
  ⟨ Read n ≤ size.l characters into buf 13 ⟩;
  mmputchars(buf, n, buffer);
  free(buf);
  o.h = 0, o.l = n;
done: return ominus(o, size);
}
```

13. ⟨ Read $n \le size.l$ characters into *buf* 13 ⟩ ≡

```
if (sfile[handle].fp ≡ stdin) {
  register unsigned char *p;
  for (p = buf, n = size.l; p < buf + n; p++) *p = stdin_chr( );
}
else {
  clearerr(sfile[handle].fp);
  n = fread(buf, 1, size.l, sfile[handle].fp);
  if (ferror(sfile[handle].fp)) {
    free(buf);
    goto done;
  }
}
```

This code is used in section 12.

14. ⟨ Subroutines 7 ⟩ +≡

```
octa mmix_fgets ARGS((unsigned int, octa, octa));
octa mmix_fgets(handle, buffer, size)
    unsigned int handle;
    octa buffer, size;
{
  char buf[256];
  register int n, s;
  register char *p;
  octa o;
  int eof = 0;
  if (¬(sfile[handle].mode & #1)) return neg_one;
  if (¬size.l ∧ ¬size.h) return neg_one;
  if (sfile[handle].mode & #8) sfile[handle].mode &= ∼#2;
  size = incr(size, −1);
  o = zero_octa;
  while (1) {
    ⟨ Read n < 256 characters into buf 15 ⟩;
    mmputchars((unsigned char *) buf, n + 1, buffer);
    o = incr(o, n);
    size = incr(size, −n);
    if ((n ∧ buf[n − 1] ≡ '\n') ∨ (¬size.l ∧ ¬size.h) ∨ eof) return o;
    buffer = incr(buffer, n);
  }
}
```

15. ⟨ Read $n < 256$ characters into *buf* 15 ⟩ ≡

```
s = 255;
if (size.l < (unsigned int) s ∧ ¬size.h) s = (int) size.l;
if (sfile[handle].fp ≡ stdin)
  for (p = buf, n = 0; n < s; ) {
    *p = stdin_chr( );
    n++;
    if (*p++ ≡ '\n') break;
  }
else {
  if (¬fgets(buf, s + 1, sfile[handle].fp)) return neg_one;
  eof = feof(sfile[handle].fp);
  for (p = buf, n = 0; n < s; ) {
    if (¬*p ∧ eof) break;
    n++;
    if (*p++ ≡ '\n') break;
  }
}
*p = '\0';
```

This code is used in section 14.

16. The routines that deal with wyde characters might need to be changed on a system that is little-endian; the author wishes good luck to whoever has to do this. `MMIX` is always big-endian, but external files prepared on random operating systems might be backwards.

```
⟨ Subroutines 7 ⟩ +≡
  octa mmix_fgetws ARGS((unsigned int, octa, octa));
  octa mmix_fgetws(handle, buffer, size)
      unsigned int handle;
      octa buffer, size;
  {
    char buf[256];
    register tetra n, s;
    register char *p;
    octa o;
    int eof = 0;
    if (¬(sfile[handle].mode & #1)) return neg_one;
    if (¬size.l ∧ ¬size.h) return neg_one;
    if (sfile[handle].mode & #8) sfile[handle].mode &= ∼#2;
    buffer.l &= −2;
    size = incr(size, −1);
    o = zero_octa;
    while (1) {
      ⟨ Read n < 128 wyde characters into buf 17 ⟩;
      mmputchars((unsigned char *) buf, 2 * n + 2, buffer);
      o = incr(o, n);
      size = incr(size, −(int) n);
      if ((n ∧ buf[2 * n − 1] ≡ '\n' ∧ buf[2 * n − 2] ≡ 0) ∨ (¬size.l ∧ ¬size.h) ∨ eof) return o;
      buffer = incr(buffer, 2 * n);
    }
  }
```

17.

```
⟨ Read n < 128 wyde characters into buf 17 ⟩ ≡
  s = 127;
  if (size.l < s ∧ ¬size.h) s = size.l;
  if (sfile[handle].fp ≡ stdin)
    for (p = buf, n = 0; n < s; ) {
      *p++ = stdin_chr( ); *p++ = stdin_chr( );
      n++;
      if (*(p − 1) ≡ '\n' ∧ *(p − 2) ≡ 0) break;
    }
  else
    for (p = buf, n = 0; n < s; ) {
      if (fread(p, 1, 2, sfile[handle].fp) ≠ 2) {
        eof = feof(sfile[handle].fp);
        if (¬eof) return neg_one;
        break;
      }
      n++, p += 2;
      if (*(p − 1) ≡ '\n' ∧ *(p − 2) ≡ 0) break;
    }
  *p = *(p + 1) = '\0';
```

This code is used in section 16.

18. ⟨ Subroutines 7 ⟩ +≡

```
octa mmix_fwrite ARGS((unsigned int, octa, octa));
octa mmix_fwrite(handle, buffer, size)
    unsigned int handle;
    octa buffer, size;
{
  char buf[256];
  register unsigned int n;
  if (¬(sfile[handle].mode & #2)) return ominus(zero_octa, size);
  if (sfile[handle].mode & #8) sfile[handle].mode &= ∼#1;
  while (1) {
    if (size.h ∨ size.l ≥ 256) n = mmgetchars(buf, 256, buffer, −1);
    else n = mmgetchars(buf, size.l, buffer, −1);
    size = incr(size, −(int) n);
    if (fwrite(buf, 1, n, sfile[handle].fp) ≠ n) return ominus(zero_octa, size);
    fflush(sfile[handle].fp);
    if (¬size.l ∧ ¬size.h) return zero_octa;
    buffer = incr(buffer, n);
  }
}
```

19. ⟨ Subroutines 7 ⟩ +≡

```
octa mmix_fputs ARGS((unsigned int, octa));
octa mmix_fputs(handle, string)
    unsigned int handle;
    octa string;
{
  char buf[256];
  register unsigned int n;
  octa o;
  o = zero_octa;
  if (¬(sfile[handle].mode & #2)) return neg_one;
  if (sfile[handle].mode & #8) sfile[handle].mode &= ∼#1;
  while (1) {
    n = mmgetchars(buf, 256, string, 0);
    if (fwrite(buf, 1, n, sfile[handle].fp) ≠ n) return neg_one;
    o = incr(o, n);
    if (n < 256) {
      fflush(sfile[handle].fp);
      return o;
    }
    string = incr(string, n);
  }
}
```

20. ⟨ Subroutines 7 ⟩ +≡

```
octa mmix_fputws ARGS((unsigned int, octa));
octa mmix_fputws(handle, string)
    unsigned int handle;
    octa string;
{
  char buf[256];
  register unsigned int n;
  octa o;
  o = zero_octa;
  if (¬(sfile[handle].mode & #2)) return neg_one;
  if (sfile[handle].mode & #8) sfile[handle].mode &= ∼#1;
  while (1) {
    n = mmgetchars(buf, 256, string, 1);
    if (fwrite(buf, 1, n, sfile[handle].fp) ≠ n) return neg_one;
    o = incr(o, n ≫ 1);
    if (n < 256) {
      fflush(sfile[handle].fp);
      return o;
    }
    string = incr(string, n);
  }
}
```

21. `#define sign_bit ((unsigned) #80000000)`

⟨ Subroutines 7 ⟩ +≡

```
octa mmix_fseek ARGS((unsigned int, octa));
octa mmix_fseek(handle, offset)
    unsigned int handle;
    octa offset;
{
  if (¬(sfile[handle].mode & #4)) return neg_one;
  if (sfile[handle].mode & #8) sfile[handle].mode = #f;
  if (offset.h & sign_bit) {
    if (offset.h ≠ #ffffffff ∨ ¬(offset.l & sign_bit)) return neg_one;
    if (fseek(sfile[handle].fp, (int) offset.l + 1, SEEK_END) ≠ 0) return neg_one;
  } else {
    if (offset.h ∨ (offset.l & sign_bit)) return neg_one;
    if (fseek(sfile[handle].fp, (int) offset.l, SEEK_SET) ≠ 0) return neg_one;
  }
  return zero_octa;
}
```

22. ⟨ Subroutines 7 ⟩ +≡

```
octa mmix_ftell ARGS((unsigned int));
octa mmix_ftell(handle)
    unsigned int handle;
{
  register long x;
  octa o;
  if (¬(sfile[handle].mode & #4)) return neg_one;
  x = ftell(sfile[handle].fp);
  if (x < 0) return neg_one;
  o.h = 0, o.l = x;
  return o;
}
```

23. One last subroutine belongs here, just in case the user has modified the standard error handle.

⟨ Subroutines 7 ⟩ +≡

```
void print_trip_warning ARGS((int, octa));
void print_trip_warning(n, loc)
    int n;
    octa loc;
{
  if (sfile[2].mode & #2)
    fprintf(sfile[2].fp, "Warning: %s at location %08x%08x\n", trip_warning[n], loc.h, loc.l);
}
```

24. ⟨ Global variables 6 ⟩ +≡

```
char *trip_warning[] = {"TRIP", "integer divide check", "integer overflow",
    "float-to-fix overflow", "invalid floating point operation",
    "floating point overflow", "floating point underflow",
    "floating point division by zero", "floating point inexact"};
```

25. Index.

__STDC__: 2.
addr: 4.
ARGS: 2, 4, 7, 8, 10, 11, 12, 14, 16, 18, 19, 20, 21, 22, 23.
big-endian versus little-endian: 16.
buf: 4, 12, 13, 14, 15, 16, 17, 18, 19, 20.
buffer: 12, 14, 16, 18.
calloc: 12.
clearerr: 13.
done: 12, 13.
eof: 14, 15, 16, 17.
f: 10.
failure: 8.
fclose: 8, 11.
feof: 15, 17.
ferror: 13.
fflush: 18, 19, 20.
fgets: 15.
FILENAME_MAX: 2, 8.
fopen: 8.
fp: 5, 7, 8, 10, 11, 13, 15, 17, 18, 19, 20, 21, 22, 23.
fprintf: 23.
fread: 13, 17.
free: 12, 13.
fseek: 21.
ftell: 22.
fwrite: 18, 19, 20.
h: 3.
handle: 8, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22.
I/O: 1.
incr: 4, 14, 16, 18, 19, 20.
input/output: 1.
l: 3.
list: 2.
little-endian versus big-endian: 16.
loc: 23.
mmgetchars: 4, 8, 18, 19, 20.
mmix_fake_stdin: 10.
mmix_fclose: 11.
mmix_fgets: 14.
mmix_fgetws: 16.
mmix_fopen: 8.
mmix_fputs: 19.
mmix_fputws: 20.
mmix_fread: 12.
mmix_fseek: 21.
mmix_ftell: 22.
mmix_fwrite: 18.
mmix_io_init: 7.
mmputchars: 4, 12, 14, 16.
mode: 5, 7, 8, 11, 12, 14, 16, 18, 19, 20, 21, 22, 23.
mode_code: 8, 9.
mode_string: 8, 9.
n: 12, 14, 16, 18, 19, 20, 23.
name: 8.
name_buf: 8.
neg_one: 4, 8, 11, 12, 14, 15, 16, 17, 19, 20, 21, 22.
o: 12, 14, 16, 19, 20, 22.
octa: 3, 4, 8, 11, 12, 14, 16, 18, 19, 20, 21, 22, 23.
offset: 21.
ominus: 4, 12, 18.
oplus: 4.
p: 13, 14, 16.
print_trip_warning: 23.
s: 14, 16.
SEEK_END: 2, 21.
SEEK_SET: 2, 21.
sfile: 6, 7, 8, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23.
sign_bit: 21.
sim_file_info: 5, 6.
size: 4, 12, 13, 14, 15, 16, 17, 18.
stderr: 7.
stdin: 7, 10, 13, 15, 17.
stdin_chr: 4, 13, 15, 17.
stdout: 7.
stop: 4.
string: 19, 20.
system dependencies: 16.
tetra: 3, 16.
trip_warning: 23, 24.
x: 22.
zero_octa: 4, 8, 11, 14, 16, 18, 19, 20, 21.

⟨ External subroutines 4 ⟩ Used in section 1.
⟨ Global variables 6, 9, 24 ⟩ Used in section 1.
⟨ Preprocessor macros 2 ⟩ Used in section 1.
⟨ Read $n < 128$ wyde characters into *buf* 17 ⟩ Used in section 16.
⟨ Read $n < 256$ characters into *buf* 15 ⟩ Used in section 14.
⟨ Read $n \le size.l$ characters into *buf* 13 ⟩ Used in section 12.
⟨ Subroutines 7, 8, 10, 11, 12, 14, 16, 18, 19, 20, 21, 22, 23 ⟩ Used in section 1.
⟨ Type definitions 3, 5 ⟩ Used in section 1.

August 21, 2014 at 15:45

MMIX-IO

	Section	Page
Introduction	1	1
Index	25	10