1. METARDB. This program uses BHT Files to store METAR reports for rapid cross-platform retrieval.

⟨ Includes 3 ⟩
⟨ Structure Definitions 6 ⟩
⟨ Global Variable Definitions 8 ⟩
⟨ Forward Declarations 49 ⟩
⟨ Function Definitions 4 ⟩
⟨ Main Program 45 ⟩

2. Useful symbolic definitions.

```
#define TRUE (1 ≡ 1)
#define FALSE (¬(TRUE))
```

3. ⟨ Includes 3 ⟩ ≡

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <signal.h>
#include <time.h>
#include <sys/time.h>
#include <errno.h>
#include <stdarg.h>
#include "bhtf.h"
```

This code is used in section 1.

4. ⟨ Function Definitions 4 ⟩ ≡

⟨ Signal Functions 50 ⟩
⟨ Utility Functions 9 ⟩
⟨ File Functions 24 ⟩
⟨ Ingest Functions 5 ⟩

This code is used in section 1.

5. ⟨ Ingest Functions 5 ⟩ ≡

⟨ Ingest Support Functions 16 ⟩

See also sections 10 and 11.

This code is used in section 4.

6. We read data from *stdin* one line at a time, but a single METAR report may be spread over several lines. To facilitate the ingest, the report may be in one of several states.

⟨ Structure Definitions 6 ⟩ ≡

```
enum states {
  Empty ← 1, RepFirstLine, RepLines, Finished
};
```

See also sections 7 and 27.

This code is used in section 1.

7. We break up the METAR report string into a few pre-parsed pieces for easier access later. The METAR header is either `"METAR"` or `"SPECI"` (five characters, specified with `METAR_HDRSIZE`). The station identifier is four characters (`STATION_STRLEN`), and the maximum length of a single METAR report is given by `MAX_REPSIZE`.

```
#define METAR_HDRSIZE 5
#define STATION_STRLEN 4
#define MAX_REPSIZE 512
```

⟨ Structure Definitions 6 ⟩ +≡

```
struct rep_t {
  char methdr[METAR_HDRSIZE + 1];
  char station[STATION_STRLEN + 1];
  char text[MAX_REPSIZE];
  time_t ztime;
  int tlen;
  enum states state;
};
```

8. ⟨ Global Variable Definitions 8 ⟩ ≡

```
char bigbuf[256];
```

See also sections 28, 40, 42, 43, and 44.

This code is used in section 1.

9. Remove leading and trailing spaces, and canonify adjacent white space merging all types into a single `'␣'`. This is all done in-place, modifying the original string s. And since we've had to process the whole string anyway, we return the cleaned-up string length as a bonus.

⟨ Utility Functions 9 ⟩ ≡

```
int string_cleaning(char *orig)
{
  char *d ← orig, *s ← orig;
  while (*s) {
    while (*s ∧ isspace(*s)) s++;    /* skip leading spaces */
    while (*s ∧ ¬isspace(*s)) *d++ ← *s++;    /* copy non-spaces */
    if (*s) {
      *d++ ← '␣'; s++;
    }
  }
  if (*(d − 1) ≡ '␣') d−−;    /* back over last trailing space */
  *d ← '\0';
  return (d − orig);
}
```

See also sections 23, 25, 26, 29, 35, 37, 38, and 39.

This code is used in section 4.

10. Write out a report to the output file.

```
⟨ Ingest Functions 5 ⟩ +≡
  void write_report(struct rep_t *r)
  {
    BHTFile outfile;
    char finalstr[MAX_REPSIZE];
    if ((outfile ← set_output(r→ztime)) ≠ Λ) {
      strcpy(finalstr, r→methdr);
      strcat(finalstr, "␣");
      strcat(finalstr, r→text);
      bht_addkey(outfile, r→station, finalstr, r→tlen + 6, BHT_VALAPPEND);
    }
    r→state ← Empty;
    r→tlen ← 0;
  }
```

11. For each line we read from *stdin*, we need to know what to do with it. The function *addtoreport*( ) keeps track of the state, and tries to do the right thing.

```
⟨ Ingest Functions 5 ⟩ +≡
  void addtoreport(char *s, struct rep_t *r)
  {
    int slen ← string_cleaning(s);
    if (¬slen) return;    /* blank lines don't cause a state change */
    switch (r→state) {
    case Empty: ⟨ Empty-state text handling 12 ⟩
      break;
    case RepFirstLine: ⟨ RepFirstLine-state handling 13 ⟩
      break;
    case RepLines: ⟨ RepLines-state handling 14 ⟩
      break;
    default: logit("Consuming␣text␣in␣unexpected␣state:␣%d\n", r→state);
    }
    if (r→state ≠ Finished) return;
    r→text[r→tlen++] ← '\n';
    r→text[r→tlen] ← '\0';
    ⟨ Parse out the station and time and write report 15 ⟩
  }
```

12. When a report is in the *Empty* state, then we should expect the next line we read to contain either `"METAR"` or `"SPECI"`. If so, we copy the string and change state to *RepFirstLine*. If not, we may have fallen out of sync with the `pqsurf` process (or it with us), in which case we let the line go by and hope a new header comes along soon.

```
⟨ Empty-state text handling 12 ⟩ ≡
  if (strcasecmp("METAR", s) ≡ 0 ∨ strcasecmp("SPECI", s) ≡ 0) {
    strcpy(r→methdr, s);
    r→state ← RepFirstLine;
  } else logit("Expecting␣header:␣received␣out-of-sync␣text␣'%s'\n", s);
```

This code is used in section 11.

13. In the *RepFirstLine* state, the new text in s gets copied to the report buffer, and then we update the length of the text in case there's more text to come. We know the report has finished if the last non-text character is an `'='`. If we don't see an `'='`, we change to the *RepLines* state to continue ingesting more text for this report. If we find that the text line is larger than anticipated for a single report, then we return to the *Empty* state and hope to re-synchronize with a new header.

```
⟨ RepFirstLine-state handling 13 ⟩ ≡
  if (slen < sizeof (r→text)) {
    strcpy(r→text, s);
    r→tlen ← slen;
    if (r→text[r→tlen − 1] ≡ '=') r→state ← Finished;
    else r→state ← RepLines;
  } else {
    r→state ← Empty;
    logit("Report␣too␣large\n");
  }
```

This code is used in section 11.

14. If we are in the *RepLines* state, then we are still waiting to read the final `'='` character. We first append a `'␣'` character to the existing report text, and then we concatenate the additional line and update the text length.

```
⟨ RepLines-state handling 14 ⟩ ≡
  if ((r→tlen + slen + 2) ≤ sizeof (r→text)) {
    r→text[r→tlen] ← '␣';
    strcpy(&(r→text[r→tlen + 1]), s);
    r→tlen += slen + 1;
    if (r→text[r→tlen − 1] ≡ '=') r→state ← Finished;
  } else {
    r→state ← Empty;
    logit("Report␣too␣large\n");
  }
```

This code is used in section 11.

15. Extract the station and the time from the report text, and if both succeed, write the final report. Otherwise, reset to the *Empty* state to sync with the next header.

```
⟨ Parse out the station and time and write report 15 ⟩ ≡
  if (¬parse_station(r→station, r→text, STATION_STRLEN)) {
    r→text[8] ← '\0';
    logit("Report␣contains␣improperly-formatted␣station␣ID:␣%s\n", r→text);
  } else if (¬parse_time(&(r→ztime), r→text + (STATION_STRLEN + 1))) {
    r→text[STATION_STRLEN + 1 + 8] ← '\0';
    logit("Report␣for␣station␣%s␣contains␣bad␣time/date:␣%s\n", r→station,
        r→text + (STATION_STRLEN + 1));
  } else write_report(r);
  r→state ← Empty;
```

This code is used in section 11.

16. Parse out the station information from the beginning of the text string s, transferring the data to the storage pointed to by d. In order to be correctly parsed, we should expect a space character to be found after *slen* characters to return `TRUE`. Otherwise, we consider the report bogus and return `FALSE`.

```
⟨ Ingest Support Functions 16 ⟩ ≡
  int parse_station(char *d, char *s, int slen)
  {
    while (slen--)
      if (isalnum(*s)) *d++ ← *s++;
      else return FALSE;
    *d ← '\0';
    if (*s ≠ '␣') return FALSE;
    else return TRUE;
  }
```

See also section 17.

This code is used in section 5.

17. Extract the time information from the report string and convert it to a UNIX timestamp. The report string has the format `'ddhhmmZ'`, and so the correct month and year need to be wrapped around it (setting seconds to 0). (This should be the closest month and year to the date and time specified.) If the format of the date/time string is not correct, return `FALSE`; otherwise, return `TRUE` to indicate successful conversion. Several automated military stations seem to return only a 4-digit time, not the date. If we sense this *shortz* format, then we set today as the current date. This strategy fails if the report time and our current time straddle midnight.

```
⟨ Ingest Support Functions 16 ⟩ +≡
  int parse_time(time_t *rtime, char *s)
  {
    int shortz, dd, hh, mm, nowmonth, nowyear;
    time_t mytime, lastmon, thismon, nextmon;
    struct tm reptime;
    shortz ← (*(s + 4) ≡ 'Z');
    ⟨ Convert time digits 18 ⟩
    ⟨ Get current time to fill in missing info 19 ⟩
    if (reptime.tm_mday ≡ dd) *rtime ← timegm(&reptime);
    else {
      ⟨ Find the month entry closest to now 21 ⟩
    }
    ⟨ Determine if date is within allowable skew 22 ⟩
    return TRUE;
  }
```

18. ⟨ Convert time digits 18 ⟩ ≡

```
if (¬shortz) {
  dd ← (*s++ − '0') * 10;
  dd += (*s++ − '0');
  if (dd < 1 ∨ dd > 31) return FALSE;
}
hh ← (*s++ − '0') * 10;
hh += (*s++ − '0');
if (hh < 0 ∨ hh > 23) return FALSE;
mm ← (*s++ − '0') * 10;
mm += (*s++ − '0');
if (mm < 0 ∨ mm > 59) return FALSE;
if (¬((*s ≡ 'Z') ∧ (*(s + 1) ≡ '␣'))) return FALSE;
```

This code is used in section 17.

19. The METAR report has the day of the report, the hour and the minute. To convert that time to a UNIX timestamp, we get the current date from the system in order to fill in the missing information.

⟨ Get current time to fill in missing info 19 ⟩ ≡

```
time(&mytime);
gmtime_r(&mytime, &reptime);
⟨ Determine day if shortz 20 ⟩
reptime.tm_hour ← hh;
reptime.tm_min ← mm;
reptime.tm_sec ← 0;
```

This code is used in section 17.

20. If the report time stamp has just the hours and minutes and not the date, then we assume that the time is from today. However, with network lag and outages, "today" gets particularly fuzzy around midnight. We attempt to compensate by detecting the most common occurrences. However, rather than back a date up to the previous month, or bump it up to the next, we prefer to throw the report away: it's too much work.

⟨ Determine day if shortz 20 ⟩ ≡

```
if (shortz) {
  if ((reptime.tm_hour ≡ 0) ∧ (hh > 18)) {
    dd ← reptime.tm_mday − 1;
    if (dd < 1) return FALSE;      /* too hard */
  } else if ((reptime.tm_hour ≡ 23) ∧ (hh < 3)) {
    dd ← reptime.tm_mday + 1;
    if (dd ≥ 28) return FALSE;      /* too hard */
  } else dd ← reptime.tm_mday;
}
```

This code is used in section 19.

21. If we get here, then the day of the report and the today's date are not the same. It may be that the date is merely a few hours on the other side of midnight from the current time. But it gets a little more complicated if midnight straddles the beginning/end of a month or even between years. To resolve the ambiguity, we choose a time conversion that results in a UNIX timestamp closest to now. We test the current month with the report date, the previous month (and possibly previous year), and the next month (and possibly next year).

```
⟨ Find the month entry closest to now 21 ⟩ ≡
  reptime.tm_mday ← dd;
  nowmonth ← reptime.tm_mon;
  nowyear ← reptime.tm_year;
  thismon ← timegm(&reptime);
  if (nowmonth ≡ 0) {
    reptime.tm_mon ← 11;
    reptime.tm_year−−;
  } else reptime.tm_mon ← nowmonth − 1;
  lastmon ← timegm(&reptime);
  reptime.tm_year ← nowyear;
  if (nowmonth ≡ 11) {
    reptime.tm_mon ← 0;
    reptime.tm_year++;
  } else reptime.tm_mon ← nowmonth + 1;
  nextmon ← timegm(&reptime);
  *rtime ← closest(mytime, lastmon, thismon, nextmon);
```

This code is used in section 17.

22. Now that we've calculated the time, we do a little quality control by figuring out if the report date is within the proper time span. In general we're more lenient with older reports than with those that predict the future.

```
⟨ Determine if date is within allowable skew 22 ⟩ ≡
  if ((*rtime > (mytime + maxforwardskew)) ∨ (*rtime < (mytime − maxbackwardskew))) return FALSE;
```

This code is used in section 17.

23. Return time value closest to the present time.

```
⟨ Utility Functions 9 ⟩ +≡
  inline time_t closest(time_t now, time_t last, time_t this, time_t next)
  {
    time_t smallest;
    int delta;
    delta ← abs(last − now);
    smallest ← last;
    if (abs(this − now) < delta) {
      delta ← abs(this − now);
      smallest ← this;
    }
    if (abs(next − now) < delta) smallest ← next;
    return smallest;
  }
```

24. Verify that each dynamic component of the path exists, or create it, as necessary.

```
⟨ File Functions 24 ⟩ ≡
  int verify_path(char *path)
  {
    char *loc ← path;
    while ((loc ← strchr(loc, '/')) ≠ Λ) {
      *loc ← '\0';
      if (mkdir(path, °775) ∧ (errno ≠ EEXIST)) {
        logit("Failed␣to␣make␣path␣component␣'%s'\n", path);
        return FALSE;
      }
      *loc++ ← '/';
    }
    return TRUE;
  }
```

See also sections 30, 33, 34, and 36.

This code is used in section 4.

25. If sufficient space is left in buffer p, then convert *val* according to the format string f, and advance p by *len* characters and reduce the space left in $*sz$. If there's not enough space left in p, then set $*sz \leftarrow -1$ so that an error may be returned.

```
⟨ Utility Functions 9 ⟩ +≡
  inline char *addformat(char *p, int *sz, int len, const char *f, int val)
  {
    if (len < *sz) {
      sprintf(p, f, val);
      *sz −= len; p += len;
    } else *sz ← −1;
    return p;
  }
```

26. Given a UNIX timestamp in *tval*, expand the dynamic component of the *path_template*, inserting time values dynamically as specified.

```
⟨ Utility Functions 9 ⟩ +≡
  int instantiate_path(char *buff, int size, char *path_template, time_t tval)
  {
    struct tm t_breakout;
    char *p ← buff, *s ← path_template;
    gmtime_r(&tval, &t_breakout);
    while (*s ∧ (size > 1)) {
      if (*s ≠ '%') {
        *p++ ← *s++; size--;
      } else {
        switch (*(++s)) {
        case 'Y': p ← addformat(p, &size, 4, "%4d", t_breakout.tm_year + 1900);
          break;
        case 'M': p ← addformat(p, &size, 2, "%02d", t_breakout.tm_mon + 1);
          break;
        case 'D': p ← addformat(p, &size, 2, "%02d", t_breakout.tm_mday);
          break;
        case 'h': p ← addformat(p, &size, 2, "%02d", t_breakout.tm_hour);
          break;
        case 'm': p ← addformat(p, &size, 2, "%02d", t_breakout.tm_min);
          break;
        default: *p++ ← '%'; *p++ ← *s; size -= 2;
        }
        if (*s) s++;
      }
    }
    if (size ≤ 0) {
      logit("Dynamic␣pathname␣expansion␣is␣too␣large\n");
      return FALSE;
    }
    *p ← '\0';
    return TRUE;
  }
```

27. We may have several BHT files open at any one time, and we need a structure to keep track of the files' statuses.

```
⟨ Structure Definitions 6 ⟩ +≡
  struct bht_t {
    char *path;
    BHTFile bfid;
    time_t lastaccess, lastflush;
  };
```

28. We create an array of these structures, up to `MAX_BHTFILES`.

```
#define MAX_BHTFILES 5
⟨ Global Variable Definitions 8 ⟩ +≡
  struct bht_t bhtfile_list[MAX_BHTFILES];
```

29. If we need to open a new file, we search the list for an empty slot. Finding none, we seek the slot with the oldest *lastaccess* time, close the file and return its resources.

```
⟨ Utility Functions 9 ⟩ +≡
  struct bht_t *get_slot(void)
  {
    struct bht_t *slot ← Λ;
    time_t oldest ← time(0) + 1000;
    int i;
    for (i ← 0; i < MAX_BHTFILES; i++) {
      if (¬bhtfile_list[i].path) return &(bhtfile_list[i]);
      if ((bhtfile_list[i].lastaccess > 0) ∧ (bhtfile_list[i].lastaccess < oldest)) {
        slot ← &(bhtfile_list[i]);
        oldest ← bhtfile_list[i].lastaccess;
      }
    }
    close_file(slot);
    return slot;
  }
```

30. Here we figure out which file we need to write to, and we return the file pointer. If we can't get the file for some reason, we return Λ instead.

```
⟨ File Functions 24 ⟩ +≡
  BHTFile set_output(time_t tval)
  {
    char pathbuf[256];
    struct bht_t *slot;
    int i;
    if (¬instantiate_path(pathbuf, sizeof (pathbuf), path_dynamic, tval)) return Λ;
    ⟨ Seek an already-opened BHT file and return if found 31 ⟩
    ⟨ Open a new BHT file and return 32 ⟩
  }
```

31. ⟨ Seek an already-opened BHT file and return if found 31 ⟩ ≡

```
  for (i ← 0; i < MAX_BHTFILES; i++) {
    if (bhtfile_list[i].path ∧ (strcmp(pathbuf, bhtfile_list[i].path) ≡ 0)) {
      bhtfile_list[i].lastaccess ← time(0);
      return bhtfile_list[i].bfid;
    }
  }
```

This code is used in section 30.

32. ⟨ Open a new BHT file and return 32 ⟩ ≡

```
if (¬verify_path(pathbuf)) return Λ;
slot ← get_slot();
slot→bfid ← bht_open(pathbuf, BHT_WRITE | BHT_CREAT, MY_KEYSIZE, MY_TABSIZE, MY_NBUFFS);
if (slot→bfid) {
  logit("Opened␣file␣'%s'\n", pathbuf);
  slot→path ← strdup(pathbuf);
  slot→lastaccess ← slot→lastflush ← time(0);
  return slot→bfid;
} else {
  logit("Failed␣to␣open␣file␣'%s'\n", pathbuf);
  return Λ;
}
```

This code is used in section 30.

33. Split the pathname template into two pieces: a static component which must pre-exist, and a dynamic component which will be created as needed.

⟨ File Functions 24 ⟩ +≡

```
void split_template(char *s, int ssz, char *d, int dsz, char *t)
{
  char *l;
  while (*t) {
    l ← t;
    while (*t ∧ (*t ≠ '%') ∧ (*t ≠ '/')) t++;
    if (*t ≡ '/') {      /* previous component is still static */
      while ((l ≤ t) ∧ (ssz-- > 0)) *s++ ← *l++;
      if (¬ssz) {
        logit("Error:␣Static␣pathname␣component␣is␣too␣large\n");
        exit(1);
      }
      t++;
    } else if (*t ≡ '%') {      /* encountered a dynamic component */
      while ((l ≤ t) ∧ (dsz-- > 0)) *d++ ← *l++;
      if (¬dsz) {
        logit("Error:␣Dynamic␣pathname␣component␣is␣too␣large\n");
        exit(1);
      }
      t++;
      break;
    }
  }
  while (*t) *d++ ← *t++;
  *s ← *d ← '\0';
}
```

34. Step through the list of BHT files and close any that haven't been updated for a while, and flush any others that may need it.

```
⟨ File Functions 24 ⟩ +≡
  void flush_files(void)
  {
    time_t nowtime ← time(0);
    int i;
    for (i ← 0; i < MAX_BHTFILES; i++)
      if (bhtfile_list[i].path) {
        if (nowtime − bhtfile_list[i].lastaccess > MAX_FILEAGE) {
          logit("Last␣access␣age␣exceeded:␣'%s'\n", bhtfile_list[i].path);
          close_file(&(bhtfile_list[i]));
        } else if (nowtime − bhtfile_list[i].lastflush > MAX_FLUSH) {
          bht_flush(bhtfile_list[i].bfid);
          bhtfile_list[i].lastflush ← nowtime;
        }
      }
  }
```

35.

```
⟨ Utility Functions 9 ⟩ +≡
  void close_file(struct bht_t *slot)
  {
    if (¬slot→path) return;
    logit("Closing␣file␣'%s'\n", slot→path);
    bht_close(slot→bfid);
    slot→bfid ← Λ;
    free(slot→path);
    slot→path ← Λ;
    slot→lastaccess ← slot→lastflush ← 0;
  }
```

36. Step through the list of BHT files and close any that may be open.

```
⟨ File Functions 24 ⟩ +≡
  void closeout_files(void)
  {
    int i;
    for (i ← 0; i < MAX_BHTFILES; i++) close_file(&(bhtfile_list[i]));
  }
```

37. We create a function for writing status and info updates to the logfile a la the GEMPAK decoders.

```
⟨ Utility Functions 9 ⟩ +≡
  void logit(const char *fmt, ...)
  {
    time_t nowtime ← time(0);
    int retval, retry ← FALSE;
    struct tm t;
    va_list ap;
    if (¬stderr) reset_logfile();
  ReTry: localtime_r(&nowtime, &t);
    retval ← fprintf(stderr, "%02d%02d%02d␣%02d:%02d:%02d␣", (t.tm_year + 1900) % 100, t.tm_mon + 1,
        t.tm_mday, t.tm_hour, t.tm_min, t.tm_sec);
    if (retval < 0) {
      reset_logfile();
      if (¬retry) {
        retry ← TRUE;
        goto ReTry;
      }
    } else {
      va_start(ap, fmt);
      retval ← vfprintf(stderr, fmt, ap);
      va_end(ap);
      if (retval < 0) reset_logfile();
    }
  }
```

38. Make sure that the file slots are properly initialized.

```
⟨ Utility Functions 9 ⟩ +≡
  void initialize_file_slots()
  {
    int i;
    for (i ← 0; i < MAX_BHTFILES; i++) {
      bhtfile_list[i].path ← Λ;
      bhtfile_list[i].bfid ← Λ;
      bhtfile_list[i].lastaccess ← bhtfile_list[i].lastflush ← 0;
    }
  }
```

39. If our log file gets closed from underneath us, we will attempt to re-open the logfile. (Perhaps this happens when log files get cycled.) This also gets called if we receive SIGHUP.

```
⟨ Utility Functions 9 ⟩ +≡
  void reset_logfile(void)
  {
    fclose(stderr);
    stderr ← fopen(logfile, "a");
    if (stderr ≡ Λ) {
      printf("Couldn't␣open␣logfile␣'%s'\n", logfile);
      exit(1);
    }
    setvbuf(stderr, (char *) Λ, _IOLBF, 0);
  }
```

40. Configuration quantities. We specify the default logfile and default output template here. The logfile default can be overridden by the `"-d"` command-line argument. The output file template can be overridden with the `"-t"` command-line argument. During program initialization, we split the output file template into a static path component and a dynamic path component, and eventually we *chdir*( ) to the static path.

```
#define DEFAULT_LOGFILE "/var/log/metardb.log"
#define DEFAULT_TEMPLATE "/rtwx/metar/%Y%M%D/%Y%M%D%h.bht"
```

⟨ Global Variable Definitions 8 ⟩ +≡

```
  char *logfile ← DEFAULT_LOGFILE;
  char *ptemplate ← DEFAULT_TEMPLATE;
  char path_static[256];
  char path_dynamic[256];
```

41. We use the station identifier as the BHT search key, so the length of this identifier is specified for `MY_KEYSIZE` when creating the BHT file. `MY_TABSIZE` sets the size of the hash table. It should be a prime number larger than the number of unique station identifiers to be stored in the file. We don't need more than the minimum number of data buffers, which is specified by `MY_NBUFFS`.

```
#define MY_KEYSIZE STATION_STRLEN
#define MY_TABSIZE 5033
#define MY_NBUFFS 2
```

42. We need a place to store interim reports.

⟨ Global Variable Definitions 8 ⟩ +≡

```
  struct rep_t report;
```

43. The value of `MAX_FLUSH` determines how often (in seconds) open BHT files will be flushed to disk so that processes reading information can get the most recent data. `MAX_FILEAGE` says how long a file will remain open without any updates, and `MAX_QUIESCENCE` determines how long to keep running without any input. We keep global variables *do_flush*, *terminate*, *quiescent*, *hupped* and *lastactivity* so that the main loop may interact with the the signal-driven functions.

```
#define MAX_FLUSH 60
#define MAX_FILEAGE (60 * 18)
#define MAX_QUIESCENCE (60 * 10)
```

⟨ Global Variable Definitions 8 ⟩ +≡

```
  int do_flush ← FALSE;
  int terminate ← FALSE;
  int quiescent ← FALSE;
  int hupped ← FALSE;
  time_t lastactivity;
```

44. As a quality control measure, we don't want to let METAR reports with bad dates or times getting in. When we parse the METAR data we check the report time to make sure it is not too far into the future (*maxforwardskew*) or too far in the past (*maxbackwardskew*). These quantities should be specified in seconds. We could be more lenient with older reports, as network or system delays could possibly result in a backlog of old reports. However, METAR reports that predict the future are probably just bad. Still, a little fudge time is allowed.

```
#define HOURS (60 * 60)
```

⟨ Global Variable Definitions 8 ⟩ +≡

```
  int maxforwardskew ← 2 * HOURS;
  int maxbackwardskew ← 24 * HOURS;
```

45. Main Program. Initialize everything, and process entries from *stdin* until the pipe is closed.

```
⟨ Main Program 45 ⟩ ≡
  int main(int argc, char *argv[])
  {
    ⟨ Argument handling 46 ⟩
    ⟨ Open logfile 47 ⟩
    logit("Start-up\n");
    ⟨ Determine static and dynamic paths 48 ⟩
    initialize_file_slots();
    report.state ← Empty;
    report.tlen ← 0;
    lastactivity ← time(0);
    setup_signals();
    while (¬feof(stdin) ∧ ¬(terminate ∨ quiescent)) {
      if (fgets(bigbuf, sizeof (bigbuf), stdin)) {
        addtoreport(bigbuf, &report);
        lastactivity ← time(0);
      }
      if (do_flush) {
        flush_files();
        do_flush ← FALSE;
      }
      if (hupped) {
        reset_logfile();
        hupped ← FALSE;
      }
    }
    if (terminate) logit("Termination␣signal␣received\n");
    else if (quiescent) logit("Inactivity␣shutdown\n");
    else logit("EOF␣detected,␣shutting␣down\n");
    closeout_files();
    exit(0);
  }
```

This code is used in section 1.

46. ⟨ Argument handling 46 ⟩ ≡

```
argc −−; argv ++;
while (argc > 0) {
  if (strcmp("-t", *argv) ≡ 0) {
    argc −−; argv ++;
    if (¬argc) {
      logit("Missing␣path␣template␣specification\n");
      exit(1);
    }
    ptemplate ← *argv;
  } else if (strcmp("-d", *argv) ≡ 0) {
    argc −−; argv ++;
    if (¬argc) {
      logit("Missing␣logfile␣name\n");
      exit(1);
    }
    logfile ← *argv;
  }
  argc −−; argv ++;
}
```

This code is used in section 45.

47. Re-open the logfile pointing at the location we have specified. Set the stream to be line-buffered.

⟨ Open logfile 47 ⟩ ≡

```
stderr ← freopen(logfile, "a", stderr);
if (stderr ≡ Λ) {
  printf("Couldn't␣re-open␣log␣file␣to␣'%s'\n", logfile);
  exit(1);
}
setvbuf(stderr, (char *) Λ, _IOLBF, 0);
```

This code is used in section 45.

48. ⟨ Determine static and dynamic paths 48 ⟩ ≡

```
split_template(path_static, sizeof (path_static), path_dynamic, sizeof (path_dynamic), ptemplate);
if (chdir(path_static)) {
  logit("Error:␣Couldn't␣chdir()␣to␣'%s'\n", path_static);
  exit(1);
}
```

This code is used in section 45.

49. ⟨ Forward Declarations 49 ⟩ ≡

```
void closeout_files(void);
void close_file(struct bht_t *);
void logit(const char *fmt, ...);
void reset_logfile(void);
```

This code is used in section 1.

50. We need a function to respond to SIGALRM signals. If MAX_QUIESCENCE > 0 and if the time since the last input exceeds that value, then set *quiescent* ← TRUE to force the main loop to exit. Also set *do_flush* ← TRUE to check for any files that need to be flushed to disk.

```
⟨ Signal Functions 50 ⟩ ≡
  void alarm_handler(int sig)
  {
    time_t now ← time(0);
    if (MAX_QUIESCENCE ∧ ((now − lastactivity) > MAX_QUIESCENCE)) quiescent ← TRUE;
    else do_flush ← TRUE;
  }
```

See also sections 51 and 52.

This code is used in section 4.

51. Pass along to the main loop that we received either SIGHUP or SIGTERM/SIGQUIT.

```
⟨ Signal Functions 50 ⟩ +≡
  void termhup_handler(int sig)
  {
    if (sig ≡ SIGHUP) hupped ← TRUE;
    else terminate ← TRUE;
  }
```

52. Now we set up the signals.

```
⟨ Signal Functions 50 ⟩ +≡
  void setup_signals(void)
  {
    struct itimerval itimer;
    struct sigaction sa;
    sigset_t mask;
    sigemptyset(&mask);
    sa.sa_mask ← mask;
    sa.sa_flags ← 0;
    ⟨ Setup ignored signals 53 ⟩
    ⟨ Setup termination signals 54 ⟩
    ⟨ Setup alarm signal 55 ⟩
  }
```

53. We want to ignore most common user-sent signals.

```
⟨ Setup ignored signals 53 ⟩ ≡
  sa.sa_handler ← SIG_IGN;
  sigaction(SIGUSR1, &sa, Λ);
  sigaction(SIGUSR2, &sa, Λ);
```

This code is used in section 52.

54. We respond differently to SIGHUP, SIGQUIT, and SIGTERM.

```
⟨ Setup termination signals 54 ⟩ ≡
  sa.sa_handler ← termhup_handler;
  sigaction(SIGHUP, &sa, Λ);
  sigaction(SIGTERM, &sa, Λ);
  sigaction(SIGQUIT, &sa, Λ);
```

This code is used in section 52.

55. We set up the alarm signal to be sent every `ALARM_INTERVAL` seconds. This determines the granularity of the check for quiescence and file-flush activities.

#define `ALARM_INTERVAL` 15

⟨ Setup alarm signal 55 ⟩ ≡

```
sa.sa_handler ← alarm_handler;
sigaction(SIGALRM, &sa, Λ);
itimer.it_value.tv_sec ← ALARM_INTERVAL;
itimer.it_value.tv_usec ← 0;
itimer.it_interval.tv_sec ← ALARM_INTERVAL;
itimer.it_interval.tv_usec ← 0;
setitimer(ITIMER_REAL, &itimer, Λ);
```

This code is used in section 52.

56. Index.

⟨ Argument handling 46 ⟩ Used in section 45.
⟨ Convert time digits 18 ⟩ Used in section 17.
⟨ Determine day if *shortz* 20 ⟩ Used in section 19.
⟨ Determine if date is within allowable skew 22 ⟩ Used in section 17.
⟨ Determine static and dynamic paths 48 ⟩ Used in section 45.
⟨ File Functions 24, 30, 33, 34, 36 ⟩ Used in section 4.
⟨ Find the month entry closest to now 21 ⟩ Used in section 17.
⟨ Forward Declarations 49 ⟩ Used in section 1.
⟨ Function Definitions 4 ⟩ Used in section 1.
⟨ Get current time to fill in missing info 19 ⟩ Used in section 17.
⟨ Global Variable Definitions 8, 28, 40, 42, 43, 44 ⟩ Used in section 1.
⟨ Includes 3 ⟩ Used in section 1.
⟨ Ingest Functions 5, 10, 11 ⟩ Used in section 4.
⟨ Ingest Support Functions 16, 17 ⟩ Used in section 5.
⟨ Main Program 45 ⟩ Used in section 1.
⟨ Open a new BHT file and return 32 ⟩ Used in section 30.
⟨ Open logfile 47 ⟩ Used in section 45.
⟨ Parse out the station and time and write report 15 ⟩ Used in section 11.
⟨ Seek an already-opened BHT file and return if found 31 ⟩ Used in section 30.
⟨ Setup alarm signal 55 ⟩ Used in section 52.
⟨ Setup ignored signals 53 ⟩ Used in section 52.
⟨ Setup termination signals 54 ⟩ Used in section 52.
⟨ Signal Functions 50, 51, 52 ⟩ Used in section 4.
⟨ Structure Definitions 6, 7, 27 ⟩ Used in section 1.
⟨ Utility Functions 9, 23, 25, 26, 29, 35, 37, 38, 39 ⟩ Used in section 4.
⟨ *Empty*-state text handling 12 ⟩ Used in section 11.
⟨ *RepFirstLine*-state handling 13 ⟩ Used in section 11.
⟨ *RepLines*-state handling 14 ⟩ Used in section 11.

METARDB: A program to store METAR Reports

(Version 0.1)

Bret D. Whissel