

# ADAPTING OPEN SOURCE SOFTWARE TO YOUR DESIGN NEEDS – A HOWTO DOCUMENT

## A CASE STUDY OF THE ADAPTION OF VIXIE CRON FOR USE AS A MACHINE CONTROLLER.

### Introduction

This document covers the motivations and means of using open source software as a basis for the development of new software projects.

The majority of development experience is often contained within an engineering company, this is after all how it makes a living, or obtained by attending academic or commercial courses. The suggestion here is that open source software contains a wealth of materials and information that can, when suitably used, supplement these approaches. There is little attached cost to acquiring this experience and the process of using and adapting technology "not invented here" acts as a valuable training tool for engineers allowing them to see examples of good practice and exposing them to new designs and ideas.

The simplest expression of the design methodology is "Why reinvent the wheel?" A significant portion of the functionality used in any new software project will have been a requirement for someone else, somewhere else. The passing of information between systems, the control of electronic switching, the storage of information and in this particular case the scheduling of time-based activity are all needs that have been fulfilled by earlier software engineers.

### Why would we wish to adapt code?

The main reasons for wishing to adapt open source software are same reasons that should drive all methodologies used in software development. These are saving development costs, increasing software quality and improving programmer skill sets. How are these ends achieved?

The saving of development time is mainly due to finding and using a suitable piece of open source code around which to base your new project. The requirements for doing this are fairly straightforward. Firstly the code should contain a reasonably large portion of the functionality required. There is after all little point in trying to re-code a

word processing package to carry out mathematical calculations. Secondly the code shouldn't contain a lot more functionality that isn't required. Again we should not be using a word processing package as our start point if we aren't interested in printing, text formatting or a user interface.

Far too much development time is expended by programmers designing "new" solutions to old problems. These solutions have to then be coded and tested at great expense. When basing designs on open source code a major time saving arises through algorithms and designs already existing. In our particular case we needed to be able to express when a scheduled program needed to be run. The cron source code already has a design embodied in it and, obviously, the algorithms and source code required to put that design into practice.

The quality of the code will be enhanced in three major ways. Firstly there will be less new code to test. By judiciously blocking in new pieces of functionality core parts of the original program will still work and therefore not require major testing. In the case of a commonly used piece of code the software will have been tested by usage by a large user base for periods of up to twenty years. Because the source code has been visible bug fixes will not depend on a single individual or company but can be made by any interested party. These changes will then have been peer-reviewed by the user community before making their way into the source code.

Unit testing by the programmer will be made easier. There is, after all, a working sample of the program around. If some new piece of functionality is not working then it can be temporarily removed and added piecemeal as an aid to finding the fix. Some open source code has a lot of debugging code written in already, this is the case in Vixie cron.

Lastly programmers will, generally speaking, be exposed to some of the best code and coding practices around. Again this is due to the peer reviewing community that comes as an extra when using open source code. These good examples should lead to better practice in the programmers own work. Programmers, as much as any other people, tend to learn by example. Few companies, except the largest, will have a sufficiently wide peer population from which the coder can learn. Some open source code is amongst the best available due, as explained above, to peer review over a period of nearly 20 years. The work involved in understanding other peoples work exercises will help the coder by showing them new ideas and exercising their lateral thinking muscles and prevent stagnation.

Finally in-house development of computer systems has often led to a situation in which only one programmer "knows" or indeed can understand a system. By using source code that many people may have worked on this risk is reduced for the portion that is not original.

## Who should adapt code?

Let us consider what is required from the programmer. Firstly he or she should know their target operating system. What services are available, how are files opened and read or written, how do programs get dates and times and how do programs access i/o ports and networking. How does the programming environment work, where is the compiler, debugger and profiler. There will be examples of the use of the environment in open source code but little instruction in how to use the toolset.

Secondly the programmer must know "C" and "C++". Almost all open source code is written in these languages. "C" is the simplest and best for hardware interfacing and most of the low level hardware interfaces are written in this language. "C++" and the STL are the best for systems work. As an example a program reading the serial port would be best written in "C" but once the data is "in" the system "C++" and the STL will allow easier programming and design of the business functionality. Having got the data from our serial port processing is more easily and safely accomplished in "C++". There are libraries that will swallow a file full of data and turn it into an email (the true end of all data processing ;-)) without the programmer having to consider memory allocations and array sizes.

Thirdly the programmer must know what is out there. He or she must have a strong understanding of how computers work, which components do each sub-task in the system and what limitations are present in the design of each part. For instance the cron time scheduling service works every minute. This is no good for real time control of an aeroplane but ideal for slow moving control of a heating system or chemical processes.

Lastly the programmer must be able to "read" code. A glance at well designed code should give a reasonable overview of the core functionality. The detail can follow as required. Some people feel trapped by layout styles. This can be overcome by the use of such tools as bcpp, indent or cb ("c beautifier") which will reformat the whitespace in the code. These may convert what appears to be gibberish into a clear vision of the original author's intentions and an understanding of which pieces of the design serve which purposes. Learning, practicing and using this skill is a vital function of any programmer who works or wants to work with other people's code.

Much has already been written about designing code for re-use. The main difference between open source and standard design re-use practice is that it is less possible to design for re-use in open source. You are probably not in charge of the design of the code which is being adapted and you are definitely not in charge of a sufficient proportion of the available source code to enforce (or encourage) a re-use policy. However Unix, and as a result Linux, has as a core part of its design philosophy the

idea of small programs being efficient at single simple tasks. These basic programs are designed to be interoperable through the use of the shell. Result data from one program can be piped into the input of another program. This is a huge advantage compared to complex systems. For example if you want to edit some text to change fred to john a simple tool is available to do just that and no more. Your text will not be transformed into another file format nor will you have to re-code the file reading routines to accept the incoming data.

## Adapting Vixie Cron to carry out simple machine control.

Cron is a command scheduling program designed to launch commands requested by root and other users. The list of commands and their intended schedule are held in configuration files called crontabs which can be created by the root user and each authorised user. Cron is launched at system boot time and then loads the schedule details held in the crontabs. At a scheduled time a new process is started under the ownership and environment of the requesting user. The root user is capable of launching a process and assigning it's own ership to another user. Output from the run process is collected and sent as an email to the process owner.

The machine control software we are writing is designed to schedule the turning on of a luminometer, allow a settling period during which the machine is allowed to warm up, take a series of readings and then turn the machine off if no further readings are required. This involves launching three different processes, a serial\_reading process which starts the machine and stores reading to a temporary file, a create\_message process which reads the temporary file and creates a message which is relayed to a central server and a process that kills the serial\_reading process when the machine is required to be turned off. The software is intended to support the standard cron schedule format and a single extra reading request at a set date and time. The schedule and the extra reading request will be loaded from a single configuration file.

The act of adapting open source code can be simply split down into two tasks; adding the new requirements and removing the bits we don't need. The differences in the two requirements will affect what changes are required to the original source. The list below shows the areas of change and also the parts that we intend to keep and use. These last parts are the gains made during the development cycle.

Differences	CRON	Machine Control System
Configuration Source	Configuration data read from multiple files. Number of files unknown (dependant on current users of the cron service)	Configuration data read from single file.
Users	Cron launches commands owned by multiple users.	The control system is configured to be controlled by a "control" user.
Commands	Any commands can be scheduled.	Only three possible commands are to be launched.

Configuration Format	The configuration file only contains cron schedules.	The configuration file contains further unrelated information used in other parts of the control system.
Email	Email of command output to the owner required.	No email required as the control system has it's own logging system.
Additions		
Configuration Format	The configuration file only contains cron schedules.	The configuration file contains further unrelated information used in other parts of the control system.
Extra Readings		Extra readings required at set times.
Schedule Shifting		Small changes to the command launching procedures to allow the schedule to reflect reading times but the machine to be started prior to those times to allow warm up.
Similarities		
Schedule Data.	The format of the schedule data.	
Schedule Options	The supported options of the schedule.	
Command Launching	The command launching procedures.	
Timing	The timing engine.	

Why should we remove redundant pieces? Basically to cut down the amount of reading required for the "next" developer. The design and algorithms used in the open source software may be more complicated than we require for our functionality. For example a piece of software may be designed to deal with multiple connections or allow use by multiple users simultaneously whereas our needs may only stretch to a single connection or single user. There are two options open; we can either prune out the excess functionality – which may turn out to be fairly complicated or leave it in where appropriate. If the software is well designed having one user is only a case (possibly a special case) of having multiple users.

Here are some examples of the adaptations made to Vixie cron. They all follow the first methodology as the task of removing the extra functionality was straightforward.

## Configuration Source

Cron loads an array of commands for individual users from user crontab files in various locations in the file system. Each file can contain multiple commands and is stored with the user's environment in a linked list. This is far more complicated than our requirements; there is no concept of having different users and all our data is kept in one file /var/ines/ines which we need to read in a customised manner. The programming activities have involved removing the parts of the original code that dealt with user files and adding new reading routines for the file. However the format of the schedule data has not changed so we can keep the part that loads the schedule.

The `load_database` routine in `database.c` used to check the system crontab file and then read the spool directory for each user file checking that the user existed in `/etc/passwd`. This has all been removed and replaced with a simple check on our configuration file.

The old code is as follows:

```
void
load_database(old_db)
    cron_db      *old_db;
{
    DIR          *dir;
    struct stat   statbuf;
    struct stat   syscron_stat;
    DIR_T        *dp;
    cron_db      new_db;
    user         *u, *nu;

    Debug(DLOAD, ("[%d] load_database()\n", getpid()))

    /* before we start loading any data, do a stat on SPOOL_DIR
     * so that if anything changes as of this moment (i.e., before we've
     * cached any of the database), we'll see the changes next time.
     */
    if (stat(SPOOL_DIR, &statbuf) < OK) {
        log_it("CRON", getpid(), "STAT FAILED", SPOOL_DIR);
        (void) exit(ERROR_EXIT);
    }

    /* track system crontab file
     */
    if (stat(SYSCRONTAB, &syscron_stat) < OK)
        syscron_stat.st_mtime = 0;

    /* if spooldir's mtime has not changed, we don't need to fiddle with
     * the database.
     *
     * Note that old_db->mtime is initialized to 0 in main(), and
     * so is guaranteed to be different than the stat() mtime the first
     * time this function is called.
     */
    if (old_db->mtime == TMAX(statbuf.st_mtime, syscron_stat.st_mtime)) {
        Debug(DLOAD, ("[%d] spool dir mtime unch, no load needed.\n",
            getpid()))
        return;
    }

    /* something's different. make a new database, moving unchanged
     * elements from the old database, reloading elements that have
     * actually changed. Whatever is left in the old database when
     * we're done is chaff -- crontabs that disappeared.
     */
    new_db.mtime = TMAX(statbuf.st_mtime, syscron_stat.st_mtime);
    new_db.head = new_db.tail = NULL;

    if (syscron_stat.st_mtime) {
        process_crontab("root", "**system*",
            SYSCRONTAB, &syscron_stat,
            &new_db, old_db);
    }

    /* we used to keep this dir open all the time, for the sake of
     * efficiency. however, we need to close it in every fork, and
     * we fork a lot more often than the mtime of the dir changes.
     */
    if (!(dir = opendir(SPOOL_DIR))) {
        log_it("CRON", getpid(), "OPENDIR FAILED", SPOOL_DIR);
        (void) exit(ERROR_EXIT);
    }
}
```

```

}

while (NULL != (dp = readdir(dir))) {
    char    fname[MAXNAMLEN+1],
           tabname[MAXNAMLEN+1];

    /* avoid file names beginning with ".". this is good
     * because we would otherwise waste two guaranteed calls
     * to getpwnam() for . and .., and also because user names
     * starting with a period are just too nasty to consider.
     */
    if (dp->d_name[0] == '.')
        continue;

    (void) strcpy(fname, dp->d_name);
    sprintf(tabname, CRON_TAB(fname));

    process_crontab(fname, fname, tabname,
                    &statbuf, &new_db, old_db);
}
closedir(dir);

/* if we don't do this, then when our children eventually call
 * getpwnam() in do_command.c's child_process to verify MAILTO=,
 * they will screw us up (and v- v).
 */
endpwent();

/* whatever's left in the old database is now junk.
 */
Debug(DLOAD, ("unlinking old database:\n"))
for (u = old_db->head; u != NULL; u = nu) {
    Debug(DLOAD, ("\t%s\n", u->name))
    nu = u->next;
    unlink_user(old_db, u);
    free_user(u);
}

/* overwrite the database control block with the new one.
 */
*old_db = new_db;
Debug(DLOAD, ("load_database is done\n"))
}

```

The adapted code is:

```

void load_database (old_db)

cron_db          *old_db;

{
    struct stat    statbuf;
    cron_db        new_db;

    Debug(DLOAD, ("[%d] load_database()\n", getpid()))

    /* before we start loading any data, do a stat on /var/ines/ines so that if
     * anything changes during execution we'll see the changes next time.
     * we're interested in st_mtime (content update) and possibly st_ctime
     * (ownership update)
     */

    if (stat ("/var/ines/ines", &statbuf) < OK) {
        log_it ("CRON", getpid(), "STAT FAILED", SPOOL_DIR);
        (void) exit (ERROR_EXIT);
    }

    /* if file's mtime has not changed, we don't need to fiddle with
     * the database.
     *
     * Note that old_db->mtime is initialized to 0 in main(), and
     * so is guaranteed to be different than the stat() mtime the first
     * time this function is called.
     */
}

```

```

*/

if (old_db->mtime == statbuf.st_mtime) {
    Debug(DLOAD, ("[%d] spool dir mtime unch, no load needed.\n", getpid()))
    return;
}

/* something's different. make a new database, moving unchanged
 * elements from the old database, reloading elements that have
 * actually changed. Whatever is left in the old database when
 * we're done is chaff -- crontabs that disappeared.
*/

new_db.mtime = statbuf.st_mtime;
new_db.gid = old_db->gid;
new_db.uid = old_db->uid;
new_db.entry = NULL;
process_crontab ("/var/tmp/ines_crondlogfile", "/var/ines/ines", &statbuf, &new_db,
old_db);

/* overwrite the database control block with the new one.
*/

*old_db = new_db;
Debug(DLOAD, ("load_database is done\n"))
}

```

## Users

Cron is designed to launch jobs owned by particular users. To this end the user, group and environment associated with the job are stored in the describing structure. This is not done in the linked list of user structures to which the linked list of job entries “belongs” but the uid and gid values are saved in each job entry element. This is done because there is a slight complication to the cron system as the system cron file allows each job to be assigned to a different user. Our program is not concerned with this functionality at all so it was decided to excise it. The easiest way to do this is to remove the structural elements concerned with the user linked list and recompile! This will show the areas in which the data is used and the code can then be altered as needed. This is not just a case of blanket deletion but requires a little thought.

The original job structure declaration is as below. The uid and gid held in the entry structure are picked up as the user crontab is read:

```

typedef struct _user {
    struct _user *next, *prev; /* links */
    char *name;
    time_t mtime; /* last modtime of crontab */
    entry *crontab; /* this person's crontab */
} user;

typedef struct _entry {
    struct _entry *next;
    uid_t uid;
    gid_t gid;
    char **envp;
    char *cmd;
    bitstr_t bit_decl(minute, MINUTE_COUNT);
    bitstr_t bit_decl(hour, HOUR_COUNT);
    bitstr_t bit_decl(dom, DOM_COUNT);
    bitstr_t bit_decl(month, MONTH_COUNT);
    bitstr_t bit_decl(dow, DOW_COUNT);
}

```

```

        int                flags;
#define DOM_STAR          0x01
#define DOW_STAR          0x02
#define WHEN_REBOOT      0x04
    } entry;

typedef struct _cron_db {
    user                *head, *tail; /* links */
    time_t              mtime;        /* last modtime on spooldir */
} cron_db;

```

Note the removal, in the new code, of the user linked list and the addition of a space to store the fixed uid and gid in the cron\_db structure. The linked list pointers in the entry structure have been removed and the three different job pointers have been added.

```

typedef struct _entry {
    time_t              extra_reading_request;
    int                ERR_collation_period;
    int                collation_period;
    int                settling_period;
    int gid, uid;
    char               **envp;
    char               *cmd;
    char               reading_identifiler[51];
    bitstr_t           bit_decl(minute, MINUTE_COUNT);
    bitstr_t           bit_decl(hour, HOUR_COUNT);
    bitstr_t           bit_decl(dom, DOM_COUNT);
    bitstr_t           bit_decl(month, MONTH_COUNT);
    bitstr_t           bit_decl(dow, DOW_COUNT);
    int                flags;
#define DOM_STAR          0x01
#define DOW_STAR          0x02
#define WHEN_REBOOT      0x04
} entry;

typedef struct _cron_db {
    int gid, uid;
    entry *entry;        /* data */
    entry *sr_entry;    /* data */
    entry *so_entry;    /* data */
    time_t mtime;        /* last modtime on /etc/ines */
} cron_db;

```

The data is first picked up into the job database when the configuration files are read and then used at the point of forking to assign the ownership of the newly created process. During the lifetime of the database the user details are used to distinguish between job entries. The original code loaded the user details either as each user crontab file was opened, the first example, or for each line in the system crontab file, the second example.

(From database.c)

```

while (NULL != (dp = readdir(dir))) {
    char    fname[MAXNAMLEN+1],
           tabname[MAXNAMLEN+1];

    /* avoid file names beginning with ".".  this is good
     * because we would otherwise waste two guaranteed calls
     * to getpwnam() for . and .., and also because user names
     * starting with a period are just too nasty to consider.
     */
    if (dp->d_name[0] == '.')
        continue;

```

```

        (void) strcpy(fname, dp->d_name);
        sprintf(tabname, CRON_TAB(fname));

        process_crontab(fname, fname, tabname,
                        &statbuf, &new_db, old_db);
    }

(From entry.c)

if (!pw) {
    char          *username = cmd;      /* temp buffer */

    Debug(DPARS, ("load_entry()...about to parse username\n"))
    ch = get_string(username, MAX_COMMAND, file, "\t");

    Debug(DPARS, ("load_entry()...got %s\n",username))
    if (ch == EOF) {
        ecode = e_cmd;
        goto eof;
    }
    pw = getpwnam(username);
    if (pw == NULL) {
        ecode = e_username;
        goto eof;
    }
    Debug(DPARS, ("load_entry()...uid %d, gid %d\n",e->uid,e->gid))
}
e->uid = pw->pw_uid;
e->gid = pw->pw_gid;

```

Our design will launch it's processes with a set user identity which will be hard coded into the job creation code so the changes will involve removing the stages associating the job record with a user and altering the checks on user identity when testing job entries. Jobs are launched as previously but the uid and gid are set at the beginning of the code and copied down to the entry structure as it is created.

```

(From cron.c)

database.entry = NULL;
database.uid = 500;
database.gid = 500;

(From database.c)

e->uid = old_db->uid;
e->gid = old_db->gid;

```

## Commands

The linked lists are removed in the derived code as there is a known set of jobs to be carried out which can be loaded into only one structure. The original structure has also been altered to add new elements required to run all three different types of job (turning the machine on, taking a reading and turning it off). The original code that checks for jobs to be run and deletes job entries can be kept fairly similar, as the parameters passed in are the same.

The original code shows the reading of the command from the configuration file.

(From entry.c)

```
/* Everything up to the next \n or EOF is part of the command...
 * too bad we don't know in advance how long it will be, since we
 * need to malloc a string for it... so, we limit it to MAX_COMMAND.
 * XXX - should use realloc().
 */
ch = get_string(cmd, MAX_COMMAND, file, "\n");
```

The new code just loads in set shows the loading of three preconfigured commands.

(From database.c)

```
Debug(DPARS, ("load_entry()...about to parse command\n"))
```

```
e->cmd = "/var/ines/bin/create_message";
```

(From entry.c)

```
log_it(log_file_name, getpid(), "LOADENTRY", file_name);
e = load_entry (file, NULL);
if (e) {
    e->uid = old_db->uid;
    e->gid = old_db->gid;
    e->reading_identifier[0] = '\0';
    new_db->entry = e;
    new_db->sr_entry = (entry *) calloc(sizeof(entry), sizeof(char));
    memcpy (new_db->sr_entry, e, sizeof (entry));
    new_db->sr_entry->cmd = "/var/ines/bin/serial_reader";
    new_db->so_entry = (entry *) calloc(sizeof(entry), sizeof(char));
    memcpy (new_db->so_entry, e, sizeof (entry));
    new_db->so_entry->cmd = "kill -SIGINT `cat /var/tmp/ines_serial_reader.pid`;";
    new_db->mtime = statbuf->st_mtime;
}
```

## Email

There was an amount of code in the original cron which was designed to capture the output of the run command and email it to the correct user. This involved redirecting the stdout and stderr output to a named pipe which the parent process could read to use as the email text. This was all removed.

## Configuration Format

The first additional section is designed to read the configuration data used in the machine control system. It has been added as a block called from the original code, instead of being added inline, as this is a good method of keeping original code separate. The style of file reading used in the original cron code has been copied to make incorporation of this section easier.

```
void skip_control_parameters (FILE *file, entry *e)
{
    int    ch, comma_count = 0, i;
    struct tm ERR_time;
    char extra_reading_request[15];
    char number[10];
```

```

memset (number, '\0', sizeof (number));

/* this covers the first of the parameters - machine_identity */

while (ch != '\n') {
    ch = get_char (file);
    if (ch == EOF)
        break;
}
ch = get_char (file);

/* loading the extra_reading_request */

i = 0;
while (ch != '\n') {
    extra_reading_request[i] = ch;
    ch = get_char (file);
    i++;
    if (ch == EOF)
        break;
}
extra_reading_request[i] = '\0';
if (strcmp (extra_reading_request, "00000000000000") == 0) {
    e->extra_reading_request = 0;
}
else {
    number[0] = extra_reading_request[0];
    number[1] = extra_reading_request[1];
    number[2] = extra_reading_request[2];
    number[3] = extra_reading_request[3];
    ERR_time.tm_year = atoi (number) - 1900;
    number[0] = extra_reading_request[4];
    number[1] = extra_reading_request[5];
    number[2] = '\0';
    ERR_time.tm_mon = atoi (number) - 1;
    number[0] = extra_reading_request[6];
    number[1] = extra_reading_request[7];
    ERR_time.tm_mday = atoi (number);
    number[0] = extra_reading_request[8];
    number[1] = extra_reading_request[9];
    ERR_time.tm_hour = atoi (number);
    number[0] = extra_reading_request[10];
    number[1] = extra_reading_request[11];
    ERR_time.tm_min = atoi (number);
    number[0] = extra_reading_request[12];
    number[1] = extra_reading_request[13];
    ERR_time.tm_sec = atoi (number);
    e->extra_reading_request = mktime (&ERR_time);
}
ch = get_char (file);

/* loading the extra_reading_request collation_period */

memset (number, '\0', sizeof (number));
i = 0;
while (ch != '\n' && i < sizeof (number)) {
    number[i] = ch;
    ch = get_char (file);
    i++;
    if (ch == EOF)
        break;
}
e->ERR_collation_period = atoi (number);
ch = get_char (file);

/* loading the settling_period */

memset (number, '\0', sizeof (number));
i = 0;
while (ch != '\n' && i < sizeof (number)) {
    number[i] = ch;
    ch = get_char (file);
    i++;
    if (ch == EOF)
        break;
}

```

```

    }
    e->settling_period = atoi (number);
    ch = get_char (file);

    /* loading the collation_period */

    memset (number, '\0', sizeof (number));
    i = 0;
    while (ch != '\n' && i < sizeof (number)) {
        number[i] = ch;
        ch = get_char (file);
        i++;
        if (ch == EOF)
            break;
    }
    e->collation_period = atoi (number);

    /* ch is now the newline before the cron setting */
}

```

## Extra Readings and Schedule Shifting

This is the area in which the main changes are made to the command launching engine. They are very high level changes to cover our requirements. All the rest of the underlying code is standard cron code.

```

/*-----
What to do at each minute tick.
(a) check if we need to turn the CFL on for an extra reading request.
(b) check if we need to turn the CFL on for a schedule point.
(c) check if we have arrived at the extra reading time.
(d) check if we have arrived a reading schedule point.
(e) check if we need to turn the CFL off.

We define the lead time as the collation period plus the settling time.
The collation and settling periods are expressed in minutes and the
collation period can be set differently for the extra readings and the
scheduled readings.

The CFL is turned on by starting the program serial_reader. A reading is
taken of the output data by running the program create_message. The only
logical requirement of create_message that needs concern us here is that
the serial_input data created by serial_reader must not be thrown away if
there is a read pending.

We also must determine if we need to turn off the CFL. This is done by
sending an interrupt signal to the serial_reader program we started.

We must determine the "reason" for our read as well so that the reading
identifier can be sent to the create_message executable.
-----*/

static void cron_tick (cron_db *db)
{
    time_t ScheduleSwitchOnTime;
    time_t ERRSwitchOnTime, ERRSwitchOffMax;
    time_t CFLSwitchOffTime, CFLSwitchOffMax;
    register entry *e;

    /* (a) check if we need to turn the CFL on for an extra reading request. */

    e = db->sr_entry;
    ERRSwitchOnTime = TargetTime + (60 * e->settling_period) + (60 * e-
>ERR_collation_period);
    if (ERRSwitchOnTime >= (time_t) (e->extra_reading_request - 29)
    && ERRSwitchOnTime <= (time_t) (e->extra_reading_request + 29)) {
        CFL_on = TRUE;
    }
}

```

```

        Debug (DSCH, ("starting serial_reader for ERR\n"));
        job_add (e);
    }

    /* (b) check if we need to turn the CFL on for a schedule point. */

    ScheduleSwitchOnTime = TargetTime + (60 * e->settlng_period) + (60 * e-
>collation_period);
    if (check_schedule_time (&ScheduleSwitchOnTime, e, TRUE) == TRUE) {
        CFL_on = TRUE;
        Debug (DSCH, ("starting serial_reader for scheduled read\n"));
        job_add (e);
    }

    /* (c) check if we have arrived at the extra reading time. */
    /* the tolerance takes up any slack between starting to process */
    /* the minute and setting the time */

    e = db->entry;
    if (TargetTime >= (time_t) (e->extra_reading_request - 30)
    && TargetTime <= (time_t) (e->extra_reading_request + 30)) {

        struct tm *errtm;

        errtm = gmtime (&(e->extra_reading_request));
        strftime (e->reading_identifler, 51, "E%y%m%d%H%M%S", errtm);
        Debug (DSCH, ("extra reading request %s\n", e->reading_identifler));
        job_add (e);
    }

    /* (d) check if we have arrived a reading schedule point. */

    if (check_schedule_time (&TargetTime, e, TRUE) == TRUE) {

        struct tm *sctm;

        sctm = gmtime (&TargetTime);
        strftime (e->reading_identifler, 51, "%M%H%d%a", sctm);
        Debug (DSCH, ("scheduled reading %s\n", e->reading_identifler));
        job_add (e);
    }

    /* (e) check if we need to turn the CFL off. */

    e = db->so_entry;
    /* loop through for the next x minutes where x is the amount of time */
    /* that is settlng_time and collation_time of either sort and check if */
    /* there should be a reading. If there should be then don't turn it off */
    /* if there won't be do - the first problem is that the create message */
    /* will be launched by forking allowing switch off to occur mid-task */

    if (CFL_on == TRUE) {

        int intending_to_switch_off = TRUE;
        time_t TestTime;

        /* this condition will fail if we've just turned the device on above */

        ERRSwitchOffMax = TargetTime + (60 * e->settlng_period) + (60 * e-
>ERR_collation_period);
        if (e->extra_reading_request != 0
        && ERRSwitchOffMax < (time_t) (e->extra_reading_request - 30)) {
            intending_to_switch_off = FALSE;
            Debug (DSCH, ("extra reading request pending now %ld next %ld
ERR %ld\n", TargetTime, ERRSwitchOffMax, e->extra_reading_request));
        }
        else {
            CFLSwitchOffMax = TargetTime + (60 * e->settlng_period) + (60 *
e->collation_period);
            TestTime = TargetTime;
            while (TestTime <= CFLSwitchOffMax) {
                if (check_schedule_time (&TestTime, e, FALSE) == TRUE) {
                    intending_to_switch_off = FALSE;
                    Debug (DSCH, ("scheduled reading pending now %ld,
next %ld\n", TargetTime, TestTime));
                }
            }
        }
    }

```

```
                break;
            }
            TestTime += 60;
        }
    }
    if (intending_to_switch_off) {
        CFL_on = FALSE;
        Debug (DSCH, ("turning off CFL\n"));
        job_add (e);
    }
}
else {
    Debug (DSCH, ("CFL not on\n"));
}
}
```

As a final word - make sure you leave in the original author's credits when you use their work to create a program in a few days that builds on possibly months or years of their work. The authors have made their code available for your reuse and this kindness should be respected.