CDMS: Services and Database

Santiago de la Fuente Lasa
Javier González García
Andrea Martínez Parra
María Patino Esteban
José Antonio López Pérez

March 2026

Yebes Observatory
P.O. Box 148, E-19080
Guadalajara, Spain

Version	Date	Author	Description
1.0	13/11/2025	Santiago de la Fuente	Initial creation of the document.
2.0	13/03/2026	Santiago de la Fuente	• Implementation of automatic calibration in each experiment through the Field System. • Implementation of new logs. • Implementation of monitoring of calibration parameters.
2.1	13/03/2026	Santiago de la Fuente	Typographical corrections.
2.2	20/03/2026	Javier González	Extension of Section 4: Reading CDMS values from the Field System.
3.0	21/03/2026	Santiago de la Fuente	Final version to be published.

Document versions.

Table of Contents

List of Figures

1 Introduction

This report details the services implemented during the startup of the Raspberry Pi that manages the CDMS, the developed Database (DB), and how their data are handled. It is important to consider that several machines are involved in this process:

- **Grafana [ydatabases.oan.es]**
- **CDMS Raspberry Pi [phasecalnoisecx.oan.es]**[1]
- **Meteomaser [v-meteomaser.oan.es]**
- **RT13m Field System [fs13m.oan.es]**

Finally, we have the **CDMS_Yj** database:

- Tables:
  - CDMS: Real-time data collected every 5 minutes.
  - CDMS_history: Data extracted from the logs generated by the CDMS Raspberry Pi.
  - CDMS_Calibration: Calibration data extracted every hour.

2 UDP Server

The code that controls the UDP server (`udpServer_3.0.py`), which receives the commands from the CDMS, is located as shown in Figure 2.1. In addition, the UDP client used for calibrating the CDMS once the service has been started (`udpClient.py`) can also be found there.

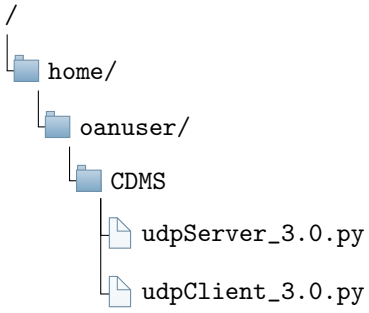

Figure 2.1: Location of `udpServer_3.0.py` [CDMS].

To automate the startup of the server, a `systemctl` service is used, placed as indicated in Figure 2.2.

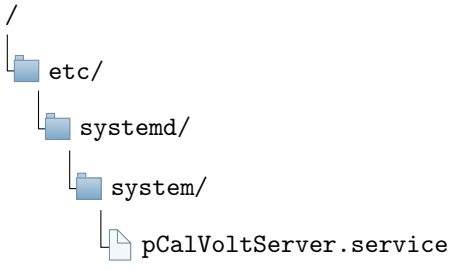

Figure 2.2: `pCalVoltServer.service` [CDMS].

[1] The domain name will be updated shortly.

The file has the following content:

```
# ---------------- pCalVoltServer.service --------------------------

[Unit]
Description=Phase Cal. GU UDP server
After=network.target
StartLimitIntervalSec=0

[Service]
Type=simple
Restart=always
RestartSec=1
User=oanuser
WorkingDirectory=/home/oanuser/CDMS
ExecStart=/usr/bin/python3 /home/oanuser/CDMS/udpServer_3.0.py

[Install]
WantedBy=multi-user.target
```

It is important to ensure that the user running the service is `oanuser` and not `root`, due to permissions. In addition, the working directory must be correctly defined.

To start the "daemon", the following commands must be executed:

```
sudo systemctl daemon-reload
sudo systemctl start pCalVoltServer.service
```

To check the status of the service:

```
sudo systemctl status pCalVoltServer.service
```

To stop or restart the service:

```
sudo systemctl stop pCalVoltServer.service
sudo systemctl restart pCalVoltServer.service
```

To inspect the output of the running code:

```
journalctl -u pCalVoltServer.service -S 2025-11-11
```

In this command, the `-S` option indicates the starting date of the log entries to be displayed; otherwise, it begins with the earliest entries. If the `-f` option is included, the latest entries will be displayed and updated automatically as new lines appear.

NOTE: It is important to launch the client to convert the Volt values into picoseconds (Calibration).

2.1 New Server Services

To operate with the Field System (FS), several functions have been implemented in the code within the file `pCalVolt.py`. In `udpServer_3.0.py`, the code is restructured to include calls to:

- `getmeas_fs`: This function returns a value to the FS depending on the *flag* `cal_flag`. If the flag is set to "True", it returns the value in picoseconds (ps). If not, it returns a 0 and a string of the form "0,invalid".
- `getcalvalues`: This function returns the current calibration values. If the CDMS is not calibrated, the values returned are `offset=0, diff=0, calFact=1`.

- `calcable`: This function already existed but has been restructured to allow extraction of calibration parameters and saving them in a log. Additional log entries were added to identify when the calibration starts and finishes, enforcing that calibration does not end until the picosecond value stabilises within the range $-5\,ps \leq x\,ps \leq 5\,ps$. Finally, the *flag* `cal_flag` was added to support the `getmeas_fs` function.

```
# ---------------- udpServer_3.0.py --------------------------

elif mesRec.lower() == "getmeas_fs":
  dataToSend = "{}".format(lastVal_fs)
elif mesRec.lower() == "getcalvalues":
  dataToSend = "Calibration: Diff (Cable - No Cable) = {} [V], Offset = {} [V],
  CalFactor = {} [ps/V]".format(diff, offset, calFactor)
elif mesRec.lower() == "calcable":
  cal_flag = True
  dataToSend = 'ACK: Performing calibration, it will take 2 min to be completed'
  logger_server.info('Message to send: {}'.format(dataToSend))
  try:
    self.sendallto(bytes(dataToSend, 'utf-8'), self.client_address, reqSocket)
  except Exception as e:
    logger_server.error('Error sending data to client: {}'.format(e))

  logger_server.info('Calibrating CDMS...')
  logger_server.info('Calibrating without cable...')
  with UDPLock:
    pcc.calibrateCable(False)
    logger_server.info('Switching relay to measure the cable...')
    pcc.setSwitch(True)

  slSec = 30
  logger_server.info('Sleeping {} seconds...'.format(slSec))
  time.sleep(slSec)
  logger_server.info('Calibrating with cable...')

  with UDPLock:
    pcc.calibrateCable(True)
    diff, offset, calFactor = pcc.calcCalibrate()
    logger_server.info('Switching relay to default position...')
    logger_server.info('Calibration: Diff (Cable - No Cable) = {} [V],
    Offset = {} [V], CalFactor = {} [ps/V]'.format(diff, offset, calFactor))
    logger_cal.info('Diff (Cable - No Cable) = {} [V], Offset = {} [V],
    CalFactor = {} [ps/V]'.format(diff, offset, calFactor))
    pcc.setSwitch(False)
    while abs(pcc.getMeasure_ps()) >= 5:
      time.sleep(0.05)
    logger_server.info('Calibration of CDMS finished...')
    dataToSend = 'ACK: Calibration finished'
    cal_flag = False
```

Furthermore, two new logs have been added, `ps` and `cal`. The `ps` log stores the picosecond values (once the CDMS is calibrated) and the `cal` log stores calibration parameters. Therefore, values in volts and picoseconds are kept in separate logs.

```
# ---------------- udpServer_3.0.py --------------------------

fileName_ps = os.path.join(os.path.normpath('/home/oanuser/logs/'),'ps.log')
fileName_cal = os.path.join(os.path.normpath('/home/oanuser/logs/'),'cal.log')

logger_ps = logging.getLogger(fileName_ps)
```

```
7   logger_ps.setLevel(logging.INFO)
8
9   logger_cal = logging.getLogger(fileName_cal)
10  logger_cal.setLevel(logging.INFO)
11
12  logHandler_cal = handlers.TimedRotatingFileHandler(fileName_cal, when='d',
13  interval=1)
14  logHandler_cal.setLevel(logging.INFO)
15  logHandler_cal.setFormatter(formatter)
16  logger_cal.addHandler(logHandler_cal)
17
18  ...
19
20  with UDPLock:
21    pcMeas = pcc.getMeasure()
22    pcMeas_ps = pcc.getMeasure_ps()
23    pcMeas_fs = pcc.getMeasure_fs(cal_flag)
24  if aux < 15:
25    aux += 1
26  else:
27    lastVal = pcMeas_ps
28    lastVal_fs = pcMeas_fs
29    logger_vol.info('{}'.format(pcMeas))
30    logger_ps.info('{}'.format(pcMeas_ps))
```

2.2 Implementation of the New Functions

For the UDP server to operate correctly, the functions must be implemented in the file `PCalVolt.py`, which is located in the same directory as the server, as shown in Figure 2.3.

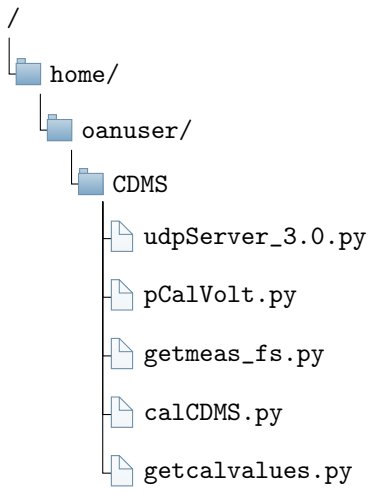

Figure 2.3: Location of `pCalVolt.py` [CDMS].

A series of simple codes have also been developed in order to test the new functionalities, as well as a simplified version of the UDP client in the file `calCDMS.py`.

The following code excerpt shows the modifications applied.

The first change made was to split the measurement process into two different functions, `getMeasure` and `getMeasure_ps`, since the intention was to distinguish between values expressed in picoseconds and those expressed in volts, allowing proper monitoring of the actual measurements obtained by the multimeter.

Additionally, the function `getMeasure_fs` has been added for use within the FS, generating "flag" strings to indicate validity and to differentiate between valid values (when the CDMS is calibrated) and invalid ones.

Finally, the `calcCalibrate` function has been modified so that it outputs the offset value, the path difference between the measurement with cable and without cable, and the conversion factor from volts to picoseconds. These values are essential in order to monitor, and potentially detect, degradation in CDMS calibration quality.

```
# ---------------- pCalVolt.py -------------------------

def calcCalibrate(self):
  self.offset = self.meas_wocable
  diff=self.meas_cable-self.meas_wocable
  self.calFactor = self.cable_cal/diff
  print("OFFSET: %f, CAL.FACTOR: %f" % (self.offset,self.calFactor))

  return diff,self.offset,self.calFactor

...

def getMeasure(self):
  volList = []
  for j in range(2):
    for i in range(4):
      v = self.read_volt()
      sleep(0.05)
      if v <= 5300:
        volList.append(v)

    final = sum(volList) / len(volList)
    self.vector = np.insert(self.vector, 0, final)
    self.vector = np.delete(self.vector, len(self.vector) - 1)
    mult = self.vector * self.filter
    GPIO.output(37, False)
    final = np.sum(mult)

  return final


def getMeasure_ps(self):
  volList = []
  for j in range(2):
    for i in range(4):
      v = self.read_volt()
      sleep(0.05)
      if v <= 5300:
        volList.append(v)

    final = sum(volList) / len(volList)
    self.vector = np.insert(self.vector, 0, final)
    self.vector = np.delete(self.vector, len(self.vector) - 1)
    mult = self.vector * self.filter
    GPIO.output(37, False)
    finalaux = np.sum(mult)
    final = (finalaux - self.offset) * (self.calFactor)

  return final


```

```
52  def getMeasure_fs(self, cal_flag):
53    if cal_flag == True:
54      output = "0,invalid"
55      return output
56    else:
57      volList = []
58      for j in range(2):
59        for i in range(4):
60          v = self.read_volt()
61          sleep(0.05)
62          if v <= 5300:
63            volList.append(v)
64
65        final = sum(volList) / len(volList)
66        self.vector = np.insert(self.vector, 0, final)
67        self.vector = np.delete(self.vector, len(self.vector) - 1)
68        mult = self.vector * self.filter
69        GPIO.output(37, False)
70        finalaux = np.sum(mult)
71        final = (finalaux - self.offset) * (self.calFactor)
72        output = "{},ok".format(final)
73
74      return output
```

3 Database: CDMS_Yj

3.1 Table: CDMS

To update the `CDMS` table in the `CDMS_Yj` database, a Python script is used, located on the `[meteomaser]` machine. This script requests data from the CDMS every 5 minutes and inserts them into the `CDMS` table within the database.

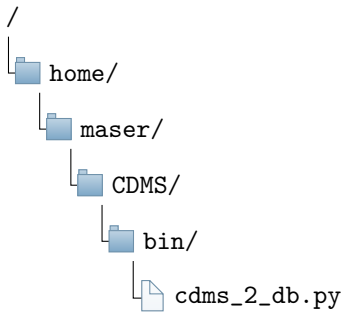

Figure 3.1: Location of the code used to request data from the CDMS and insert them into the database [meteomaser].

The `cdms_2_db.py` script is as follows:

```
1   # ---------------- cdms_2_db.py --------------------------
2   import socket
3   import sys
4   import time
5
6   import mysql.connector
7   from mysql.connector import Error
8
9   # Create a UDP socket
10  sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
```

```
11
12  server_address = ('*******', *******)
13
14  MYSQL_HOST = "localhost"
15  MYSQL_DB = "CDMS_Yj"
16  MYSQL_USERID = "*******"
17  MYSQL_PASSWD = "*******"
18  MYSQL_TABLE1 = "CDMS"
19
20  CDMSOK = False
21
22  DEBUG = True
23
24  try:
25    # Send data
26    message = 'getmeas'
27    if DEBUG:
28      print('Sending "%s"' % message)
29    try:
30      sent = sock.sendto(message.encode(), server_address)
31      # Receive response
32      if DEBUG:
33        print('Waiting to receive')
34      time.sleep(1)
35      data, server = sock.recvfrom(4096)
36      if DEBUG:
37        print('received "%s"' % data.decode())
38      CDMS_value = float(data.decode())
39      CDMSOK = True
40    except Exception as e:
41      if DEBUG:
42        print(f"CDMS error {e}")
43      CDMSOK = False
44
45    try:
46      connection = mysql.connector.connect(
47        host=MYSQL_HOST,
48        user=MYSQL_USERID,
49        password=MYSQL_PASSWD,
50        database=MYSQL_DB
51      )
52
53      if connection.is_connected():
54        cursor = connection.cursor()
55
56        if CDMSOK:
57        query = f"""
58          INSERT INTO {MYSQL_TABLE1} (ps)
59          VALUES ({CDMS_value:4.12f})
60        """
61          if DEBUG:
62            print(f"\nQuery for CDMS:\n{query}")
63          cursor.execute(query)
64
65        if DEBUG:
66          print("------------------------------------------------------------")
67
68      connection.commit()
69
70    except Exception as e:
```

```
71        print(f"MySQL error {e}")
72
73      finally:
74        if 'cursor' in locals() and cursor:
75          cursor.close()
76        if 'connection' in locals() and connection.is_connected():
77          connection.close()
78
79    finally:
80      print('closing socket')
81      sock.close()
```

NOTE: It is important to highlight that, for Grafana to access the database, a user (`grafana`) must be created in MySQL/MariaDB with the hostname `ydatabases.oan.es`, and the `50-server.cnf` file must be modified, as shown in Figure 3.2.

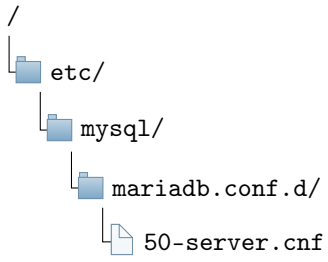

Figure 3.2: MariaDB server configuration [meteomaser].

Inside this file, the `bind-address` line must be modified and replaced with:

```
# -------------- 50-server.cnf ------------------
...
bind-address = 0.0.0.0
...
```

3.2 Table: CDMS_history

To extract the data from the logs generated daily on the CDMS, the files shown in Figure 3.3 must be checked, retrieving the data corresponding to the desired dates. Every day, the contents of the `vol.log` file are transferred into these rotated log files, following the date structure.

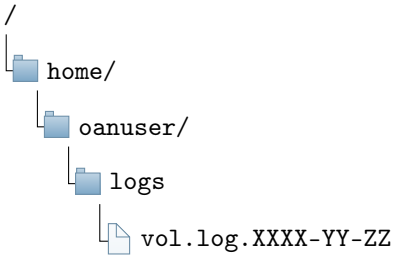

Figure 3.3: Logs on the CDMS [CDMS].

The code that extracts the data from the logs is the following:

```
# ---------------------- cdms_history_2_mysql.py -------------------------------

import os
import glob
import mysql.connector
from datetime import datetime, timedelta

# --- CONFIGURATION ---
LOG_PATH = "/home/oanuser/logs"    # Path where vol.log.YYYY-MM-DD files are located
BATCH_SIZE = 10000                 # Number of rows per batch
MARGIN_HOURS = 24                  # Safety margin in hours to cover incomplete data
DEBUG = True

# --- SQL ---
CREATE_TABLE = """
CREATE TABLE IF NOT EXISTS CDMS_history (
  id INT AUTO_INCREMENT PRIMARY KEY,
  ts DATETIME(3) UNIQUE,
  ps DOUBLE(16,12)
)
"""

INSERT_SQL = """
INSERT INTO CDMS_history (ts, ps)
VALUES (%s, %s)
ON DUPLICATE KEY UPDATE ps = VALUES(ps)
"""

SELECT_LAST_TS = "SELECT MAX(ts) FROM CDMS_history"


# --- FUNCTIONS ---
def parse_line(line):
  """Parse a line like '2025-11-04 09:28:06,698 44.8707100615836'"""
  try:
    date_str, time_millis, ps_str = line.strip().split()
    time_part, millis = time_millis.split(',')
    ts = datetime.strptime(f"{date_str} {time_part}.{millis}",
    "%Y-%m-%d %H:%M:%S.%f")
    ps = float(ps_str)
    return ts, ps
  except Exception:
    return None, None


def get_last_timestamp(conn):
  """Get the last timestamp stored in the database"""
  cur = conn.cursor()
  cur.execute(SELECT_LAST_TS)
  result = cur.fetchone()
  cur.close()
  return result[0] if result and result[0] else None


def get_local_logs(min_date):
  """Get all log files with date >= min_date."""
  all_files = sorted(glob.glob(os.path.join(LOG_PATH, "vol.log.*")))
  filtered_files = []
```

```
    for f in all_files:
      try:
        fname_date_str = f.split('.')[-1]
        fname_date = datetime.strptime(fname_date_str, "%Y-%m-%d").date()
        if fname_date >= min_date:
          filtered_files.append(f)
      except Exception:
        continue
    if DEBUG:
      print(f"Found {len(filtered_files)} log files to process.")
    return filtered_files


  def insert_into_db():
    """Insert new lines into MySQL in batches, only processing necessary logs"""
    try:
      conn = mysql.connector.connect(
        host="*******",
        user="*******",
        password="*******",
        database="CDMS_Yj",
        connection_timeout=10,
        autocommit=False
      )
    except mysql.connector.Error as e:
      print(f"Error connecting to MySQL: {e}")
      return

    cur = conn.cursor()
    cur.execute(CREATE_TABLE)
    conn.commit()

    last_ts = get_last_timestamp(conn)
    if last_ts:
      original_last_ts = last_ts
      last_ts = last_ts - timedelta(hours=MARGIN_HOURS)
      min_date = last_ts.date()
      if DEBUG:
        print(f"Last record in database: {original_last_ts}")
        print(f"Applying safety margin of {MARGIN_HOURS} hours ->
        filtering from {last_ts}")
    else:
      min_date = datetime.min.date()
      if DEBUG:
        print("No previous records found in the database.")

    log_files = get_local_logs(min_date)

    batch = []
    added = 0
    skipped = 0
    ignored = 0

    for file in log_files:
      if DEBUG:
        print(f"Processing log file: {os.path.basename(file)}")

      try:
        with open(file, "r") as f:
          lines = f.readlines()
```

```
119       except Exception as e:
120         if DEBUG:
121           print(f"Warning: could not read {file}: {e}")
122         continue
123
124       for line in lines:
125         ts, ps = parse_line(line)
126         if ts and ps is not None:
127           if last_ts and ts <= last_ts:
128             ignored += 1
129             continue
130
131         batch.append((ts, ps))
132         if len(batch) >= BATCH_SIZE:
133           try:
134             cur.executemany(INSERT_SQL, batch)
135             conn.commit()
136             added += len(batch)
137             if DEBUG:
138               print(f"Inserted batch of {len(batch)} new rows.")
139           except mysql.connector.Error as e:
140             print(f"Error inserting batch: {e}")
141           batch.clear()
142         else:
143           skipped += 1
144
145     if batch:
146       try:
147         cur.executemany(INSERT_SQL, batch)
148         conn.commit()
149         added += len(batch)
150         if DEBUG:
151           print(f"Inserted final batch of {len(batch)} new rows.")
152       except mysql.connector.Error as e:
153         print(f"Error inserting final batch: {e}")
154
155     cur.close()
156     conn.close()
157
158     if DEBUG:
159       print("\nSummary:")
160       print(f"   {added} new lines inserted or updated.")
161       if ignored:
162         print(f"   {ignored} older lines ignored (already in DB or within margin).")
163       if skipped:
164         print(f"   {skipped} lines skipped due to invalid format.")
165
166
167   def main():
168     insert_into_db()
169
170
171   if __name__ == "__main__":
172     main()
```

Additionally, the program is located as shown in Figure 3.4.

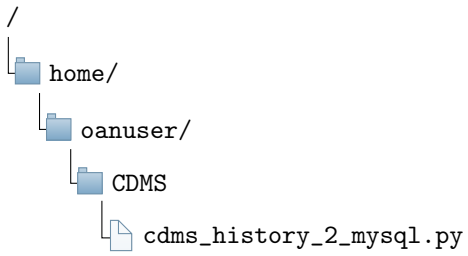

Figure 3.4: Location of the `cdms_history_2_mysql.py` program [CDMS].

This program extracts all data from the logs, checking the timestamp of the last entry stored in the `CDMS_Yj` table, and retrieving the previous 24 hours of records as a safety margin. If the entries already exist in the database, they are ignored, and only the new data are inserted. To improve performance, the program performs batch insertions of 10,000 rows.

To run the program automatically once per day, the system uses the files `cdms_history.service` and `cdms_history.timer`. A systemd timer is used to execute the script every day at `09:30 UTC`.

Both files are stored as shown in Figure 3.5.

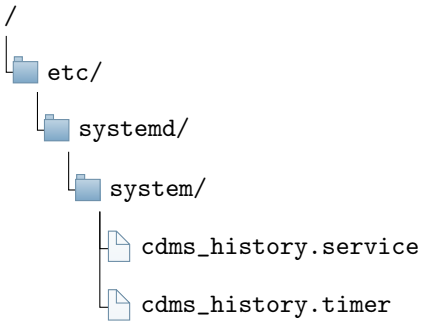

Figure 3.5: `cdms_history.service` [CDMS].

```
# ---------------------- cdms_history.service -------------------------------
[Unit]
Description=Run cdms_history_2_mysql.py every day

[Service]
Type=oneshot
ExecStart=/usr/bin/python3 /home/oanuser/CDMS/cdms_history_2_mysql.py
User=oanuser
```

```
# ---------------------- cdms_history.timer -------------------------------
[Unit]
Description=Timer to run cdms_history_2_mysql.py everyday

[Timer]
OnCalendar=*-*-* 09:30:00
Persistent=true

[Install]
WantedBy=timers.target
```

Once these files have been created, the timer `cdms_history.timer` must be enabled using `systemctl`.

To check its status and the next scheduled execution time, use:

```
systemctl list-timers cdms_history.timer
```

3.3 Table: CDMS_Calibration

Finally, the `CDMS_Calibration` table stores the exported calibration data. These data are exported every hour, since they only change when a new calibration is performed. The exported values include: Offset (measured without cable) [in V], the path difference between measurements with and without cable [in V], and the conversion factor from volts to picoseconds [in ps/V].

To update this table in the `CDMS_Yj` database, a Python script is used, located on the `[meteomaser]` machine.

This script requests the calibration data every hour and inserts them into the `CDMS_Calibration` table.

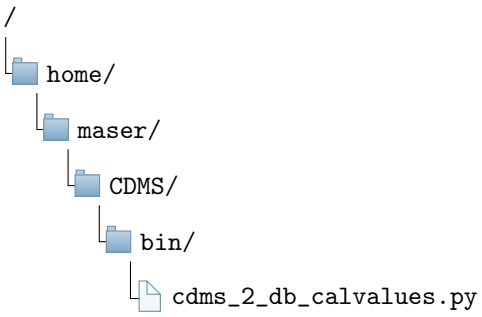

Figure 3.6: Code that requests calibration data from the CDMS and inserts them into the database [meteomaser].

The Python code implementing this behaviour is the following:

```
# ----------------------- cdms_2_db_calvalues.py ------------------------------

import socket
import sys
import time
import re

import mysql.connector
from mysql.connector import Error

# Create a UDP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

server_address = ('*******', *******)

MYSQL_HOST = "localhost"
MYSQL_DB = "*******"
MYSQL_USERID = "*******"
MYSQL_PASSWD = "*******"
MYSQL_TABLE2 = "CDMS_Calibration"

CDMSOK = False

DEBUG = True

try:
```

```
    # Send data
    message = 'getcalvalues'
    if DEBUG:
      print('Sending "%s"' % message)
    try:
      sent = sock.sendto(message.encode(), server_address)
      # Receive response
      if DEBUG:
        print('Waiting to receive')
      time.sleep(1)
      data, server = sock.recvfrom(4096)
      if DEBUG:
        print('received "%s"' % data.decode())

      patron = r"Diff .*?= ([\d.]+).*?Offset = ([\d.]+).*?CalFactor = ([\d.]+)"

      match = re.search(patron,  data.decode())

      if match:
        diff = float(match.group(1))
        offset = float(match.group(2))
        calfactor = float(match.group(3))
        if DEBUG:
          print(diff, offset, calfactor)

      CDMSOK = True
    except Exception as e:
      if DEBUG:
        print(f"CDMS error {e}")
      CDMSOK = False

    try:
      connection = mysql.connector.connect(
        host=MYSQL_HOST,
        user=MYSQL_USERID,
        password=MYSQL_PASSWD,
        database=MYSQL_DB
      )

      if connection.is_connected():
        cursor = connection.cursor()

        create_table_query = f"""
        CREATE TABLE IF NOT EXISTS {MYSQL_TABLE2} (
          id INT AUTO_INCREMENT PRIMARY KEY,
          diff DECIMAL(12,10),
          offset_value DECIMAL(12,10),
          calfactor DECIMAL(20,10),
          ts TIMESTAMP DEFAULT CURRENT_TIMESTAMP
          )
        """

        cursor.execute(create_table_query)
        connection.commit()

      if CDMSOK:
        query = f"""
          INSERT INTO {MYSQL_TABLE2} (diff, offset_value, calfactor)
          VALUES (%s, %s, %s)
        """
```

```
87
88          valores = (diff, offset, calfactor)
89
90          if DEBUG:
91            print("\nQuery for CDMS:")
92            print(query)
93            print("Valores:", valores)
94
95          cursor.execute(query, valores)
96
97        if DEBUG:
98          print("------------------------------------------------------------")
99
100       connection.commit()
101
102     except Exception as e:
103       print(f"MySQL error {e}")
104
105     finally:
106       if 'cursor' in locals() and cursor:
107         cursor.close()
108       if 'connection' in locals() and connection.is_connected():
109         connection.close()
110
111   finally:
112     print('closing socket')
113     sock.close()
```

An additional service, `cdms_calibration.service`, has been created to manage the automatic execution of the above script every hour.

Its location is shown in Figure 3.7.

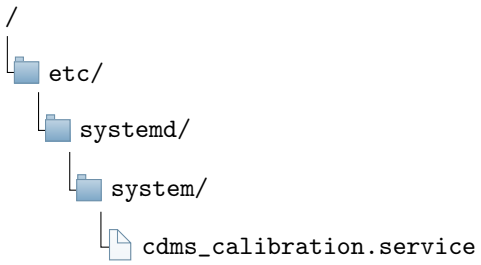

Figure 3.7: `cdms_calibration.service` [CDMS].

“

```
1   # ---------------------- cdms_calibration.service -------------------------------
2   [Unit]
3   Description=CDMS data acquisition service
4   After=network.target
5
6   [Service]
7   Type=simple
8   ExecStart=/usr/bin/python3 /home/maser/CDMS/cdms_2_db_calvalues.py
9   Restart=always
10  RestartSec=1h
11  User=maser
12  StandardOutput=append:/var/log/cdms.log
13  StandardError=append:/var/log/cdms.err
```

```
14
15  [Install]
16  WantedBy=multi-user.target
```

4 Automatic Calibration of the CDMS and Readings from the Field System

The content of this section affects exclusively the installation of the Field System for the RT13m antenna. Up to now, CDMS readings from the Field System (FS) have been carried out using the station command `cable`. This command, defined in `/usr2/control/stcmd.ctl`, corresponds to case 3 within `stqkr` and requires the argument `=getmeas`, which is sent directly to the instrument. The instrument receives this string and returns the value in picoseconds, which is recorded in the FS log under the label `/CDMS/`. This command appears in several station procedures, for example:

```
1  define fase 25351100527x
2  cable=getmeas
3  enddef
```

The measurement appears in the FS log like this:

```
1  2025.351.00:00:00.64&fase/cable=getmeas
2  2025.351.00:00:00.64/CDMS/-431.091278
```

Ordinarily, the operator does not perform calibrations of the instrument; this task falls to the engineering team, who evaluate it periodically. However, recent studies have shown that the calibration drifts over time, causing significant measurement errors within a matter of weeks. For this reason, it has been decided to include an automatic procedure that performs the calibration immediately before starting a new VLBI experiment, making use of the new functions added to the CDMS UDP server described earlier.

To implement this behaviour, the file `station.prc` (Figure 4.1) — the station's main procedure file — has been modified. Specifically, the procedure `exper_init` has been updated to include the call to `calcable`. This procedure is always executed at the start of a schedule and typically does not have its own experiment-specific definition. The function used to read cable delay measurements has also been updated: `cable` is now called without requiring the argument `=getmeas`.

```
1  # ----------------------- station.prc -------------------------------
2  define exper_initi 26071171656
3  sched_initi
4  calcable
5  ...
6  define setup4xSBC 00000000000x0x
7  calcable
8  ...
9  define faseinit 00000000000x
10 cable
11 ...
12 define faselong 00000000000x
13 cable
14 calcable
15 ...
16 define sched_end 26071182948x
17 cable
```

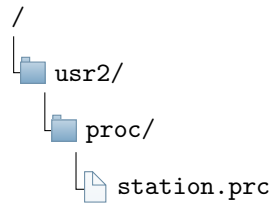

Figure 4.1: Procedure file for the RT13m station [fs13m.oan.es].

To implement the `calcable` function and the modification to `cable` within the FS, changes were made to the files `udpcable.c` (Figure 4.2) and `stcmd.ctl`.

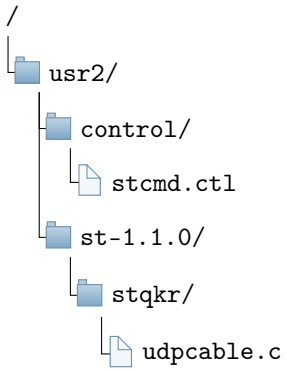

Figure 4.2: CDMS command-control files for RT13m [fs13m.oan.es].

In the `stcmd.ctl` file, the `cable` command has been updated so that the corresponding `itask` is now 2 in the `udpcable.c` code, and `calcable` corresponds to `itask` 3. In addition, a new command `testcable` has been added with `itask` 4, which prints the strings returned by the `getmeas_fs` function, including the validity flags `<invalid>` and `<ok>`. This command generates records such as:

```
2026.062.21:08:02.87/cdms2/13.334631279290829,ok
2026.063.00:31:26.67/cdms2/0.00,invalid
```

The motivation for creating a separate command with its own log output is compatibility with existing FS-log reading software used in VLBI data analysis pipelines. Analysts frequently use their own scripts to extract CDMS-related information from FS logs. The `cable` command continues to print only the value in picoseconds, ensuring compatibility with programs such as nuSolve. Additionally, each FS procedure now includes a call to `testcable` immediately after the `cable` call, so that both types of records are generated. This allows analysts to evaluate the usefulness of the new logs without altering existing workflows.

In parallel, a coordinated effort has begun to inform IVS analysis centres about this new type of log entry.

```
# ---------------------- stcmd.ctl -------------------------------

*********STATION SPECIFIC COMMANDS**************
*COMMAND     SEG SBPA BO
wx           stq 00101 01 FFFFFFFFFFFF
calnoise     stq 00201 01 FFFFFFFFFFFF
phasecal     stq 00202 01 FFFFFFFFFFFF
calmm        stq 00203 01 FFFFFFFFFFFF
*gpibnetw     stq 00301 01 FFFFFFFFFFFF
```

```
cable        stq 00302 01 FFFFFFFFFFFF
calcable     stq 00303 01 FFFFFFFFFFFF
testcable    stq 00304 01 FFFFFFFFFFFF
gps-stm      stq 00401 01 FFFFFFFFFFFF
newlo        stq 00501 01 FFFFFFFFFFFF
attrx        stq 00601 01 FFFFFFFFFFFF
pmodel       stq 01001 01 FFFFFFFFFFFF
```

```
// ---------------------- udp_cable.c -------------------------------

const char *cmd_getmeas = "getmeas_fs";
const char *cmd_calibrate = "calcable";
...
switch (itask){
  case 2: // send getmeas_fs and log ONLY value
    strcpy(inbuf,cmd_getmeas);
    txbytes = strlen(inbuf);
    if (sendto(sock, inbuf, txbytes, 0, (struct sockaddr *) &socaddin,
    sizeof(socaddin)) != txbytes )
    {
      //printf("send failed\n");
      goto error;
    }
    rxbytes = 0;
    // recv() returns 0, if server has closed the connection
    rxbytes = recvfrom(sock, outbuf, spaceleft, 0, (struct sockaddr *) &socaddin,
    &socaddin_len);
    if (rxbytes < 0){
      //printf("send failed\n");
      goto error;
    }else if (rxbytes == 0){
      printf("Connection closed\n");
    }else{
      char *coma = strchr(outbuf,',');
      if (coma != NULL){
        outbuf[rxbytes]='\0';
        *coma ='\0';
        char *resp1 = outbuf;
        char *resp2 = coma+1;
        shm_addr->cablev=atof(resp1);
        sprintf(output,"CDMS/%.2f",shm_addr->cablev);

      }else{
        goto error;
      }
    }
    break;
  case 3: // send calcable
    strcpy(inbuf,cmd_calibrate);
    txbytes = strlen(inbuf);
    if (sendto(sock, inbuf, txbytes, 0, (struct sockaddr *) &socaddin,
    sizeof(socaddin)) != txbytes )
    {
      //printf("send failed\n");
      goto error;
    }
    rxbytes = 0;
    // recv() returns 0, if server has closed the connection
    rxbytes = recvfrom(sock, outbuf, spaceleft, 0, (struct sockaddr *) &socaddin,
```

```
52        &socaddin_len);
53        if (rxbytes < 0){
54          //printf("send failed\n");
55          goto error;
56        }else if (rxbytes == 0){
57          printf("Connection closed\n");
58        }else{
59          outbuf[rxbytes]='\0';
60          sprintf(output,"CAL_CDMS/%s",outbuf);
61        }
62        break;
63      case 4: // send getmeas_fs and log BOTH value and status
64        strcpy(inbuf,cmd_getmeas);
65        txbytes = strlen(inbuf);
66        if (sendto(sock, inbuf, txbytes, 0, (struct sockaddr *) &socaddin,
67        sizeof(socaddin)) != txbytes )
68        {
69          //printf("send failed\n");
70          goto error;
71        }
72        rxbytes = 0;
73        // recv() returns 0, if server has closed the connection
74        rxbytes = recvfrom(sock, outbuf, spaceleft, 0, (struct sockaddr *) &socaddin,
75        &socaddin_len);
76        if (rxbytes < 0){
77          //printf("send failed\n");
78          goto error;
79        }else if (rxbytes == 0){
80          printf("Connection closed\n");
81        }else{
82          char *coma = strchr(outbuf,',');
83          if (coma != NULL){
84            outbuf[rxbytes]='\0';
85            *coma ='\0';
86            char *resp1 = outbuf;
87            char *resp2 = coma+1;
88            shm_addr->cablev=atof(resp1);
89            sprintf(output,"TEST_CDMS/%.2f,%s",shm_addr->cablev,resp2);
90
91          }else{
92            goto error;
93          }
94        }
95        break;
```

5 Grafana

In Grafana [ydatabases.oan.es], the CDMS plots can be viewed under the section `RT13m → CDMS`. Figure 5.1 shows two graphs representing data from both database tables. Additionally, a widget is available that allows easy monitoring of the latest real-time CDMS value.

In the real-time graph, email alerts have been configured to trigger whenever the values fall outside the range $-700\,\text{ps} \leq \text{delay} \leq 700\,\text{ps}$. If an alert is triggered—either due to an out-of-range value or because the query server becomes unreachable—Grafana records both the trigger time and the recovery time on the graph using blue vertical lines.

Finally, new plots have been added to monitor the CDMS calibration parameters, as shown in Figure 5.2.

These plots allow us to verify whether the CDMS calibration is behaving correctly or whether degradation appears in the measurement of the cable-path difference.

Figure 5.1: CDMS plots in Grafana.

Figure 5.2: CDMS calibration plots in Grafana.