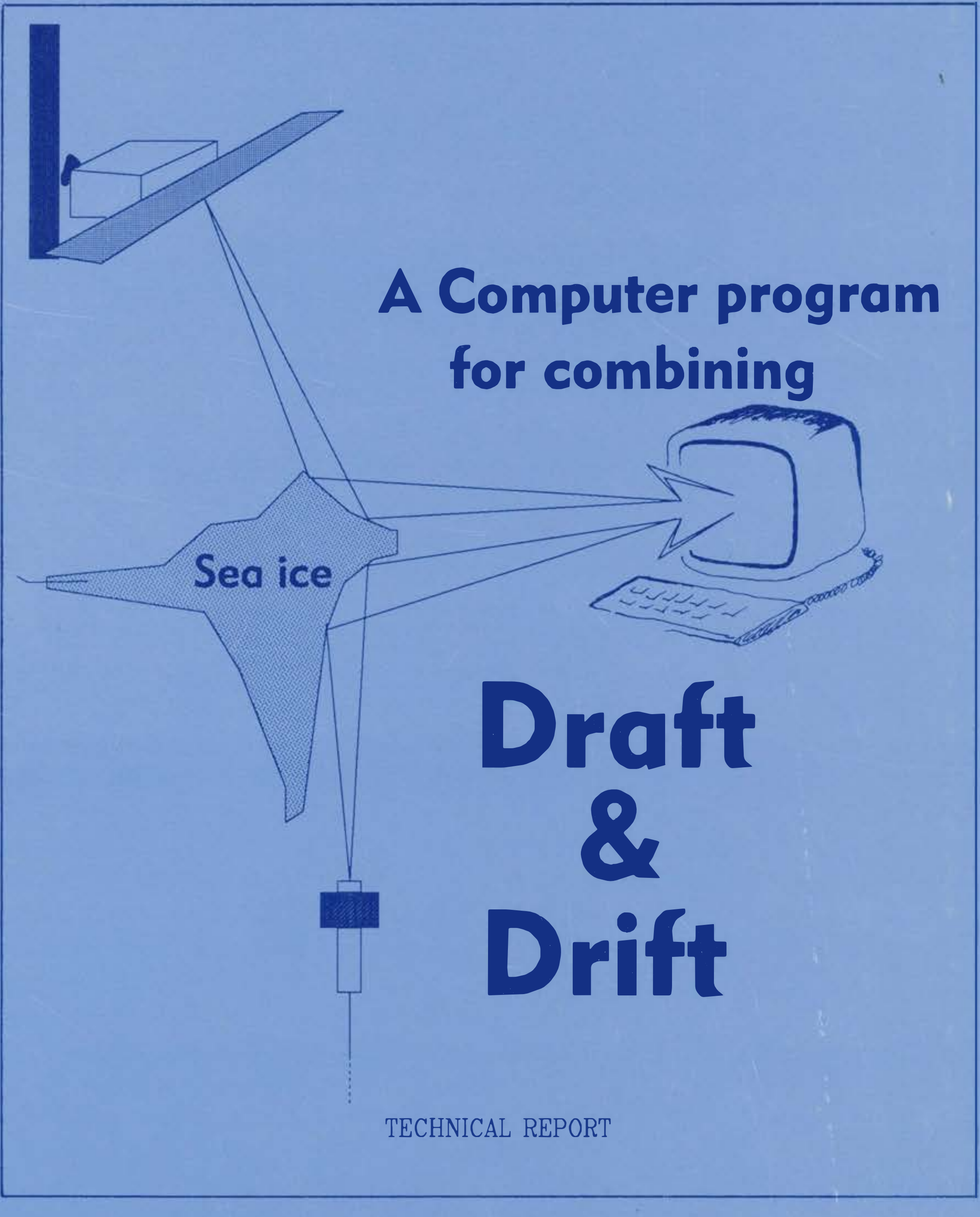
NORSK POLARINSTITUTT

RAPPORTSERIE

NR. 76 - OSLO 1992

KJELL B - E OLSSON

A Computer program for combining Draft & Drift



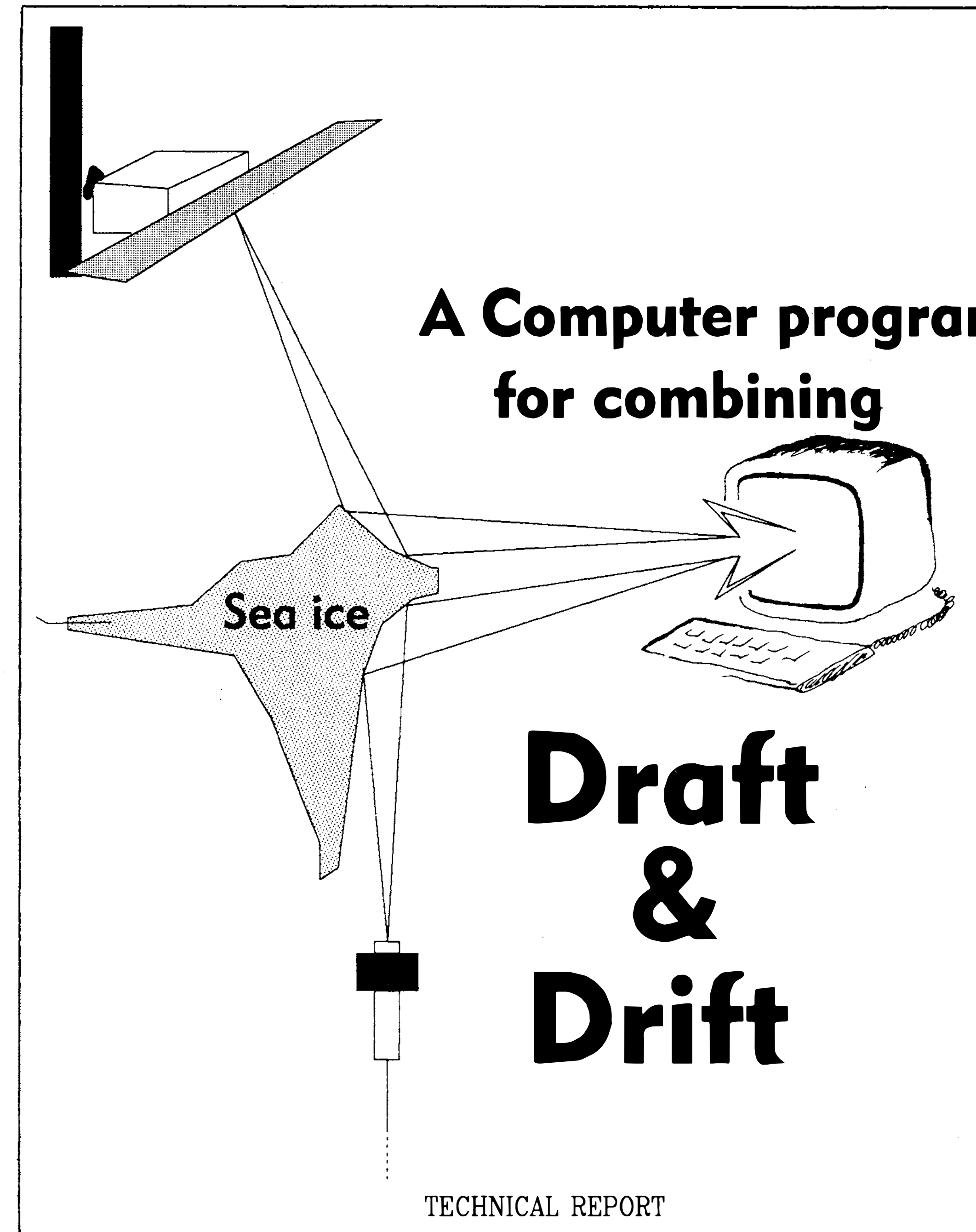
TECHNICAL REPORT

NORSK POLARINSTITUTT

RAPPORTSERIE

NR. 76 - OSLO 1992

KJELL B - E OLSSON



A Computer program for combining Draft & Drift

TECHNICAL REPORT

ISBN 82-7666-035-5
Printed May 1992

A Computer Program for Combining Sea Ice Draft and Drift

Kjell B-E Olsson

Abstract. On a suggestion from the Norwegian Polar Research Institute (NP) a moored upward looking sonar has been developed by the Christian Michelsen Institute in Bergen. Ice draft time series from this instrument combined with ice drift data from a pair of satellite images can give an estimate of the ice volume transport in a local area. This report describes a computer program developed at NP for this combination. The program is written in C and tested on a sonar series from 1987-88 at 75 03.4 N, 12 09.2 W, in the Greenland Sea. It is combined with drift data from NOAA AVHRR images, but the program is also intended for SAR and radar data. Also included are draft diagrams at 24 h resolution or less for the cloud free periods during the time period 22 of June 1987 to 20 of June 1988, extracted with the program.

Note
The prgram listing included in this report does not quite correspond to the description given in the text. This is because the printing was postponed until the stabilized version was available. There are no major functional differences between the previous, described version, called 0.6, and the listed version, called 1.0. The draft part of version 0.6 has been restructured, modularized and stabilized by John Thingstad, University of Oslo. Better validation has also been added to the program. Program structures for both versions are listed in Appendix C, to aid users of the former version.

In the latter version a fourth order Runge-Kutta method is used to find the distance to the ice subsurface. This is described at the end of Appendix A1. A logarithmic function is still used for the temperature by depth but is not iterated as before. A separate temperature is computed for each subdepth interval.

All times are converted to time passed since 1 of January 1980 and version 1.0 is valid from 1980 until 2080. GMT time is used. Storage requirements for floating additions in double precission limits the number of days in an interval to about ten days. Changing of month is not handled in version 1.0.
A control sorting of the ULS-file is advisable before the search for the appropriate time interval. The ULS-files are not always completely sorted, just almost sorted, i. e. a few records in each file may have the wrong sequence.
A temporary file is used for computing the statistics. Some minor changes have been made to the statistics file layout. Dynamic allocation is not used as before.

In version 1.0 air pressure is used so that the 00Z air pressure is used from 00:00 to 12:00 and the 12Z air pressure is used from 12:00 to 24:00.
The header block of parameters is excluded. Depths for the temperature measurements are set in the air pressure routine. Constant parameters and ULS dependent variables are set in the respective subroutines/functions, mostly as static constants.
Validation of all input is introduced and the program is adjusted for PC use.

In other aspects version 1.0 corresponds to version 0.6 described in the report text.

Contents

Background

The polar sea ice cover has a fundamental influence on the global climate and on the global hydrological system. It hemispherical and regional influence are even greater. The ice has a noticeable impact on the increasing human activities in the polar regions, making it therefore of interest to monitor, and also forecast, the sea ice extent, movements and general dynamics in time and space.

Previous to the aerial era of Man, the sea ice was monitored from ships on northern routes and from landbased weather stations. With the evolution of aeroplane and satellite image technology new means for ice monitoring over remote and vast areas emerged. In good weather the ice extent and conditions can be mapped. Using repeated satellite coverages with some time lapse, ice displacement can be extracted. This technique has been utilized by several researchers, using different kinds of images. Computations of the areal ice transport have been performed for various time intervals and locations. Assessments of transported ice volume, based on these values, have also been reported (Vinje & Finnekåsa 1986).

To estimate the transported ice volume it is necessary to know the draft of the ice and preferably also the spatial distribution of the draft. This information combined with ice drift data gives a measure of transported ice volume for a given time interval. Draft measurements have thus far been rather sparse. Profiles of the ice bottom topography have been recorded with upward looking sonars, ULS, on submarines. The coverage is poor both in time and in space. Local soundings of the ice bottom topography have been performed using a rotated scanning ULS lowered through a hole in the ice (Johnsen 1989).

By combining continuous ULS measurements with ice drift data, one obtains a record of the ice volume transport. This can be of interest in many places, but particularly in the Greenland Sea. The major part of the Arctic ice leaves the Arctic Basin through the Fram Strait via the East Greenland Current, EGC. By monitoring the ice volume transport in the EGC it is possible to get an indicator of the climatological state in the Arctic, with some time lag. The ice transport by the Transpolar Drift Stream, TPDS, takes about four years from the Sibirian coast to the Greenland Sea (Wiese 1922), rendering a good indicator of the climatic conditions in the Arctic Ocean over a period of some years.

The Norwegian Polar Research Institute suggested the use of ULS attached to the top of moorings in deep water. The first operational ULS-bouy constructed according to this principle was developed by the Christian Michelsen Institute in Bergen, Norway. Its technical specifications are given in Table 1. A successful deployment was made in 1987-88 in the Greenland sea at 75 03.4 degrees N, 12 09.2 degrees W. It was attached on top of a current meter mooring of the Alfred Wegener Institute for Polar and Marine Research, Bremerhaven, Germany. The mooring anchor was at a depth of 1242 m and the nominal depth of the ULS was 45 m below the seasurface. It was deployed at 0930 UT on 22 of June 1987 and retrieved 1800 UT on 20 of June 1988. The recording interval was set to four minutes which gave a time series of about 130 000 samples.

In combining ULS ice draft series with drift data extracted from pairs of satellite images, it must be kept in mind that the time resolution is very different. The draft measurements are stored every fourth minute while the drift, or displacement, extractions are made for a time interval of several hours, often days. However, the drift of the ice is far less variable with time compared with the rapidly changing draft.

The changing period, i. e. the "inertia" of the processes or the frequency, are quite different. Ice draft is mainly a spatially varying feature, and even a minor relative motion can cause dramatic changes in its value. Drift of sea ice is a more time-dependent feature and the frequency of change is lower and the transitions are not so abrupt. Particularly in the EGC the drift is very systematic, due to the general strength and stability of the current. Its direction is usually to the south/southwest and the displacement some ten(s of) metres per minute. Its variance is influenced by changing wind and eddy effects. Examples showing various displacement fields in the Greenland Sea can be found in Dech (1990).

The drift data is supposed to be computed based on ERS-1 images. The ERS-1 satellite was, however, not in orbit during most of this project. NOAA AVHRR data can be a good complement to ERS-1 data also when ERS-1 is operative. A computer program for mapping of ice drift, ice concentration and ice edge from ERS-1 images has been developed in cooperation between The Norwegian Computing Center and The Norwegian Polar Research Institute. The program was intended for ERS-1, but can probably be used for AVHRR images as well, if the different pixel sizes are considered (Schistad et al. 1990). The basic algorithm (Zhang 1990) is utilized successfully on AVHRR images.
NOAA AVHRR scenes received at Tromso Satellite Station, TSS, were selected for all periods during the ULS year which were cloud free over the area of interest for at least two days, rendering 25 images and giving a total of 14 periods. The time intervals varied between about ten hours and five days. A list of the images is found in Ttable 2.

Objectives

The main project is defined as "Long Time Monitoring of Ice Mass Transport (Phase I)". The objective of this subproject is to develope software for the combination of ULS ice draft series with ice drift vector data. The ice drift vector data should be based on AVHRR(Advanced Very High Resolution Radiometer) or SAR (Synthetic Apperture Radar) data. It was also an objective to arrive at an assessment of the feasability of this combination.

Developed software

A computer program has been developed for the combination of draft data from the ULS and ice displacement data from a pair of satellite images. This is a first attempt to accomplish a combination over an arbitrary time interval.

The program is mainly developed in two units, one for draft computations, handling and averaging, and one for spatial weighting of drift vector values. The draft computation is described in Appendix A1 and the algorithm is found in Appendix A2. The drift averaging is described in Appendix B1 and the algorithm in Appendix B2. The combination of the two mean values, draft and drift, gives a parameter that actually represents vertical area. Some statistics of the draft, ULS depth and drift components, are also computed.

The program is implemented in ANSI C in a workstation environment. The operating system has been ULTRIX, a member of the UNIX family. Some test and development versions have preceded the current version. After the development of subunits, several working subparts have been implemented at the Norwegian Polar Research Institute, including the two major subunits in working condition.Seven weeks were used for development of the draft part and ten days for the drift part. This also includes learning C to a usable level.

Operational depth	20-70 m
Sonar beam width	5.0 deg
Operational acoustic frequency	300 kHz
Resolution	0.1 m
Pressure transducer range	20-70 m
Resolution	0.02 m
Tilt resolution(xy)	1 deg
Data recording interval	4 min
Data recorder	Sea Data Model 610
Storing capacity (300 ft cassette)	550 days
Total length of instrument	1.70 m
Diameter of float	0.55 m
Diameter of cylinder	0.16 m
Weight in air, without float	58 kg
Total weight in air with float	79 kg
Net bouyancy in seawater	55 kg

Table 1. Technical specifications for the ULS, CMI ES-300_II, deployed 1987-1988.

Date		Satellite	Starting time
6 Nov	1987	NOAA-10	10:36:05
7 Nov	1987	NOAA-9	12:21:39
16 Nov	1987	NOAA-10	10:19:54
20 Nov	1987	NOAA-9	13:23:00
23 Nov	1987	NOAA-10	09:26:52
4 Dec	1987	NOAA-10	10:28:00
6 Dec	1987	NOAA-9	13:51:22
15 Dec	1987	NOAA-9	12:12:31
16 Dec	1987	NOAA-10	09:26:50
27 Dec	1987	NOAA-10	08:47:40
30 Dec	1987	NOAA-9	12:51:30
12 Jan	1988	NOAA-10	09:40:00
14 Jan	1988	NOAA-9	10:09:31
20 Jan	1988	NOAA-10	10:06:20
21 Jan	1988	NOAA-9	12:14:30
26 Mar	1988	NOAA-9	13:55:00
27 Mar	1988	NOAA-9	05:25:00
27 Mar	1988	NOAA-10	09:08:59
20 Apr	1988	NOAA-10	10:27:40
25 Apr	1988	NOAA-9	13:31:00
12 May	1988	NOAA-10	10:49:10
13 May	1988	NOAA-9	13:36:20
9 Jun	1988	NOAA-10	10:39:31
10 Jun	1988	NOAA-9	05:10:45
10 Jun	1988	NOAA-10	10:17:29

Table 2. The selected NOAA AVHRR scenes. Each scene is exactly four minutes long, i. e. the finishing time is exactly four minutes after the starting time given in the table.

Usage of the developed program

Needed programs

To run the present program the following files must be available somewhere with a known path:

draftdrift.c	the source program
nrutil.h	routine for dynamical allocation of real value arrays
irutil.h	as nrutil, but for integer values, also including some error handling
utm.h	routine for conversion utm <---> global coordinates
uls*yymm*.dat	**one** ULS data file. *yymm* is year and month, for example uls8801.dat
ltr*yyyy*.dat	**one** air pressure data file. *yyyy* is the first and last year, for example ltr8788.dat
tempcurr.dat	**one** file with shallow temperature- and current data (v1281.dat used)
temp.dat	**one** file with deeper temperature data (v981.dat used)
drift.dat	**one** drift data file. The name is optional, given by the user.

The name of air pressure and temperature data files are defined in the program header unit.
The various data formats are described in appendix D.

At present the compilation can be done directly, if the paths defined in the header unit are correct. No extra linking is necessary. A final version may use a Makefile (Feldman ????) command to handle the compilation and linking.

Using the program

When starting the program the user is first asked for the year, month, and day of the first and last day of the timeinterval of interest. This should be given in the y2d format, year, month, day, without separators, for example 19880609 or 880609. If desired, a particaular time of the day can be given for start and end time. This should be given in 24 h time, separated with a colon, for example 09:15, 23:35. If data for the wanted time period is missing, an error message is given.

After the draft computations are finished, the user is asked for the limit value for negative draft that should be included. Any value can be given, but reasonable values could be 0.0, -0.2 or -0.5. To exclude newfrozen ice a value of + 0.5 can be used.

The user is then asked if (s)he wants the statistics printed to a file. If data of the position is missing, the user is asked for position values. The position values are

normally collected from the header of the file with shallow temperature and current data. The next question is if the user wants the computed data tabulated on a file. If the answer is yes, choice questions follow. Draft is always included, but ULSdepth, time (Julian day and minute number in day), and record markings are optional. Header text for the file is optional. If header text is wanted, a slightly different format is used. The data is tabulated in ASCII-format in one to five columns. The column length is at present not always correct, which can give some zero records at the end. This is because some samples may be missing in the ULS file.
The table file can be used for transferring data to some other data handling package, for example a statistical or a graphical package.

When the draft part is finished the user is asked if there are any drift data files available. If one answers “no” the program is terminated, for example when the interest is merely for draft data or if a displacement data file is missing. If the answer is “yes”, the user is asked for the name of the displacement data file.

If the file is found and successfully opened, the user is promted to give the frame form, the center coordinates, and the number of frames wanted for the displacement averaging. Frames can be circular or rectangular. Coordinates should be in decimal latitude and longitude. At present one to five frames can be used.This first interrogation is followed by questions about the dimensions and weights for each frame. Dimensions should be given in km and weights as multiplication factors. The frames should be concentric, like an onion, with no border line intersections.

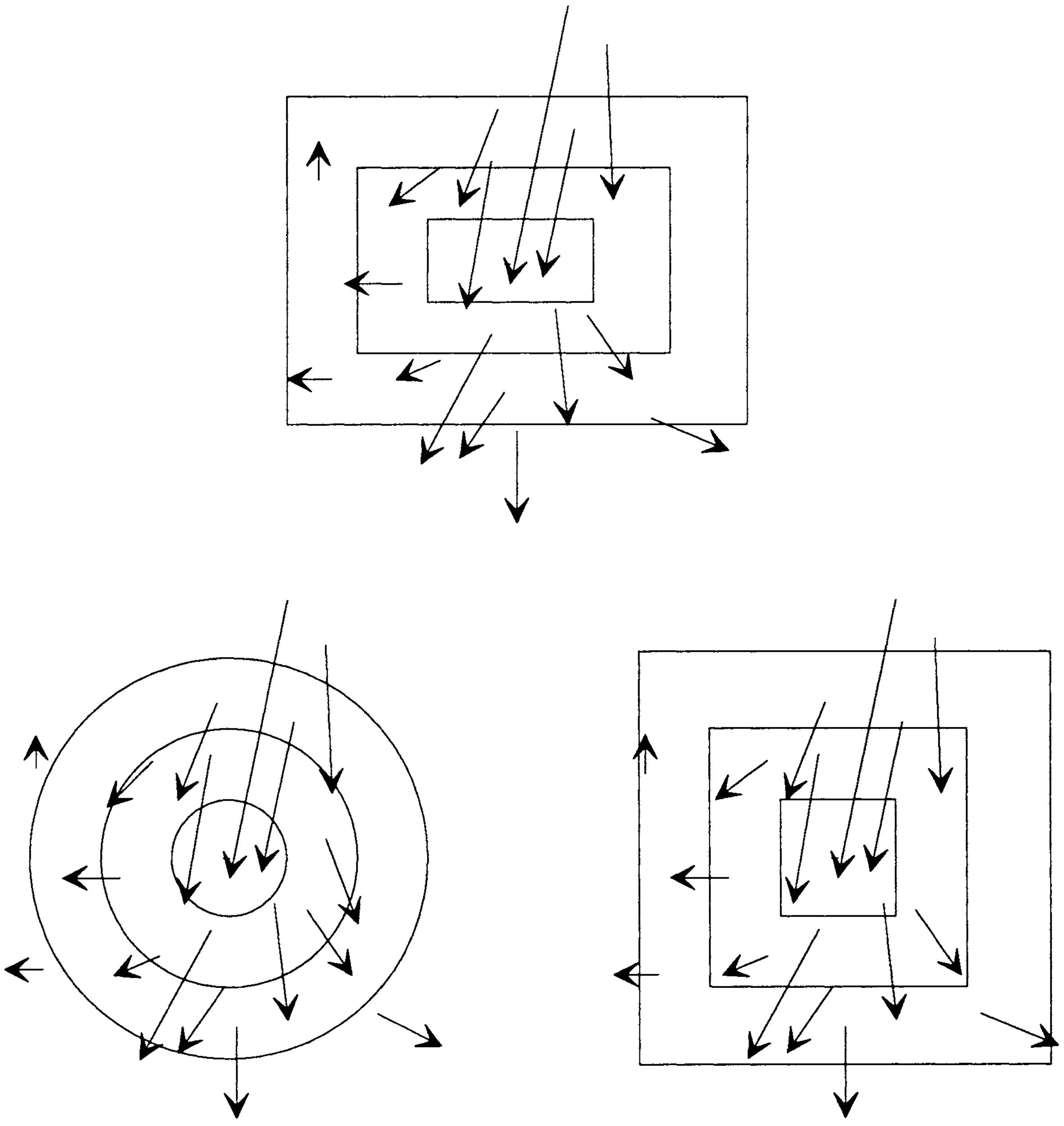


Fig. 1 Examples of frames for the vector field averaging. Rectangular frames are also represented as sqares. Some differences can be seen, depending on frame form.

Results

The developed software is the main result of the present project. Mainly because of technical reason, files missing etc, the displacement vector files from the received AVHRR images haven't been completed. Draft data have been extracted for 13 of the 14 cloud free periods covered by the AVHRR image pairs. Technical causes, probably file format error, prevented the period 26 - 27 March to be plotted. Fig. 2 shows an example of one of the statistical printouts from the program and Fig. 3 shows an example of the table files.

The extracted data have been plotted using the program package StatView on a Macintoch II. These diagrams are appended to this rapport as a presentation of the data (Appendix F). The longer periods are split in 24 h units, from start time (Table 2), and they have all been plotted using the same vertical scale. The boundaries are set by the the extreme values, considering all the periods. Both draft and ULS-depth have been converted to metres below sealevel. Periods much shorter than 24 h have been extended in the horisontal direction to improve the time resolution and better expose the ice features. All draft values are computed using the temperature value from the 72 m temperature sensor. More compressed descriptive statistics, computed using StatView, are presented for each of the plotted periods in Appendix G.

For one of the cloud free periods, 12-13 May 1988, one displacement and drift vector file have been available from Zhangs measurements (described in Zhang 1988) and vector field examples have been given in Zhang (1989)). The average drift computed from these vectors has been used for a plot of vertical area. The drift is recomputed to yield drift per sample, which is multiplied with the draft to give vertical area. It has been plotted with positive values, [m^2]. It is primarely intended as an example of the combination of draft and drift and not so much to give exact numerical values. The time is cut at 24 h. Descriptive statistics are computed as for the draft diagrams. Also added is a diagram of the frequency distribution.

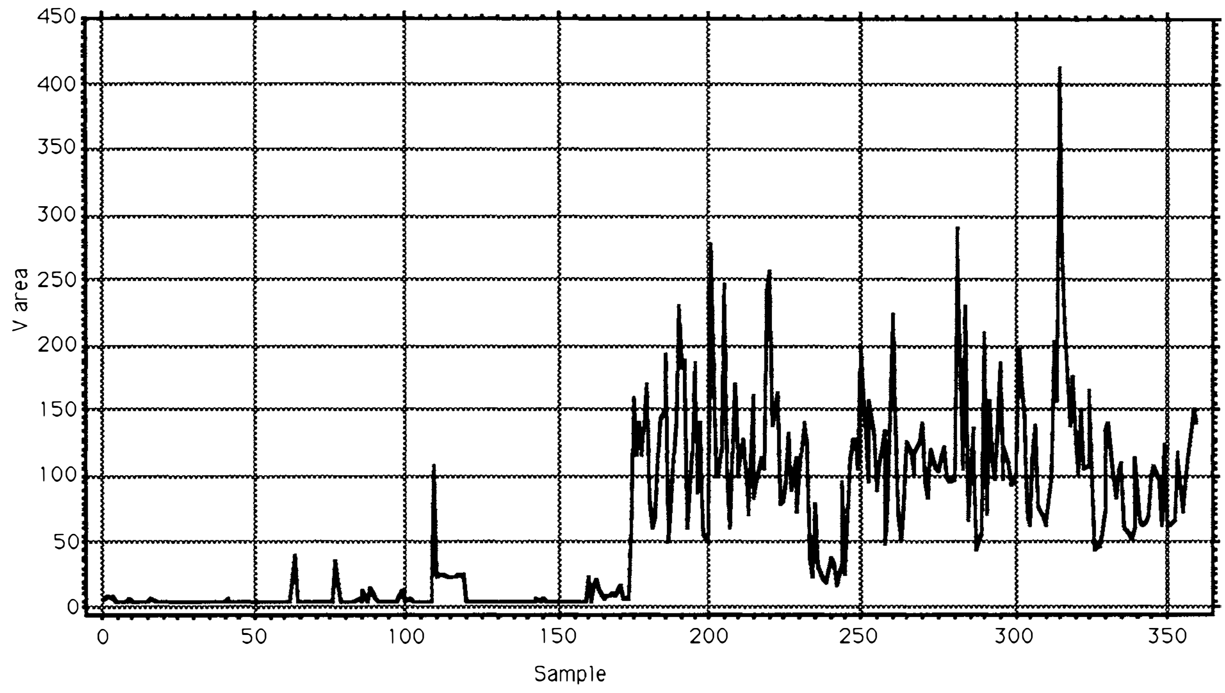
```
Descriptive parameters for ULS measurement.
Printed Fri Jul 19 03:52:45 1991

The period is from 88:020:0608 to 88:021:0736

Latitude:  75  3.2   Longitude:   -12  -11.2    Waterdepth: 1242 m

Temp. at 101 m          Temp. at 72 m        Computed temp.   Soundspeed
-0.151                  -1.236               -1.236           1440.224
-0.321                  -1.466               -1.466           1439.962

=============================================================
The data are DRAFT data.

MEAN of this period is:  2.330   based on      388 values

54  samples were marked as aproximate
3   samples were regarded out of reasonable limits and therefore not included

The MAXIMUM value is:   12.593   and the MINIMUM value is:   0.171

Values less than  -0.20 were not included in the statistical computations.

The MEAN DEVIATION is:   1.649   and the STANDARD DEVIATION is:   2.098

The VARIANCE is:   4.403

The SKEWNESS is:   1.302   and the KURTOSIS is:   2.017


=============================================================
The data are ULS DEPTH data.

MEAN of this period is:   44.523   based on    391 values

The number of approximate values as above.

The MAXIMUM value is:   47.699   and the MINIMUM value is:   42.659

The MEAN DEVIATION is:   1.138  and the STANDARD DEVIATION is:   1.430

The VARIANCE is:   2.046

The SKEWNESS is:   0.840  and the KURTOSIS is:   -0.473
```

Fig. 2 Example of a description file. Later drift values and current will be included. (Temperature from 72 m used.)

Julian day	Minute number	Draft	Depth of ULS
2	24	1.391	44.094
2	28	4.320	44.094
2	32	3.508	44.074
2	36	4.576	44.055
2	40	4.835	44.094
2	44	4.999	44.055
2	48	8.277	44.035

Fig. 3 Example of a table file with header.

Combined draft and drift.

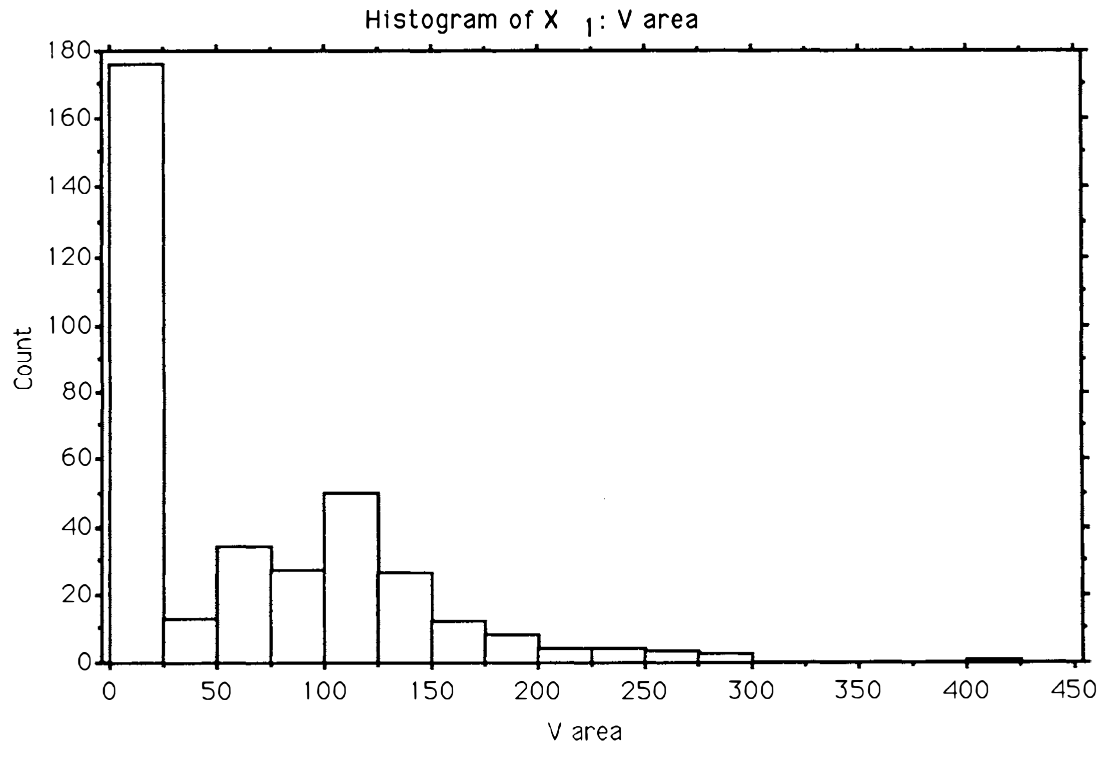


Period 880512, vertical area 24 h.

X_1 : V area

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
62,576	67,328	3,548	4533,02	107,593	360
Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
3,526	412,078	408,552	22527,52	3037046,165	0

Descriptive statistics for the above 24 h.



Frequency distribution for the vertical area of the period 880512.

Discussion

Former, usually areal, ice transport has been computed. Several authors have emphasized the importance of computing ice volume transport, for example Moritz (1988). The sea ice draft and drift program presented here is an effort to accomplish such a task. Even though the method has not yet (Sep. 1991) been completed for operational conditions, it looks very promising and it shows one possible way to approach the problem. The program for the draft evaluation also provides a tool for analysing the draft data of an arbitrary time period. It can be used to produce plots and descriptive parameters for various time resolutions. This can be of interest, for example in the study of ice bottom topography, ice floe dynamics etc.

There are several reasons C has been chosen as the programming language. Most programming today is performed in C. By using C under UNIX (in this case ULTRIX), the program becomes more portable. This is of special interest considering the remarkable rise of interest for the ULS ice thickness series, which the program is primarely intended for. Handling large data sets and images can be considerably more effective using pointers to addresses, compared to direct addressing of positions. The good possibility of allocating dynamically in C saves space and makes the program more computer economic.

All important variables are gathered in a header block. They are defined using the C preprocessor. Thus maintenance and upgrading become very easy to perform. The header block also gives a good overview. The paths to all used in-files are defined in this way. Some alternative paths are also listed. These are at present commented out, for example to the original dataset. This makes it very easy to change direction to different locations of the source files. Paths which are not included can easily be added. Calibration variables are also defined in the header block. This is a clear advantage when the program should be used for another ULS.

Global variables are used to a relatively large extent. This is not as clear as passing variables, but easier to implement with limited experience in pointer usage. This will probably be changed to some degree in a later version. One advantage with passing variables (or in C usually pointers to them) is that the entering status of a subroutine can easily be checked using debugger commands. The debugger Dbx is used in this project.

The main sources of error for the draft computation is the lack of salinity values, the precision and averaging of the temperature, damping of the sound energy in the water column, and the averaging effect of the sonar beam footprint. These sources cause draft deviations in the dm scale, or normally less than a dm. The variations have in some aspects a systematic effect, but they cancel out to some degree in the draft averaging. Wave effects can have an influence on the marginal ice zone. Wave effects and footprint effects are illustrated in Vinje (1991). These can also be seen in a few of the segments of the 14-day diagrams published in the data report (Vinje & Berge 1989).

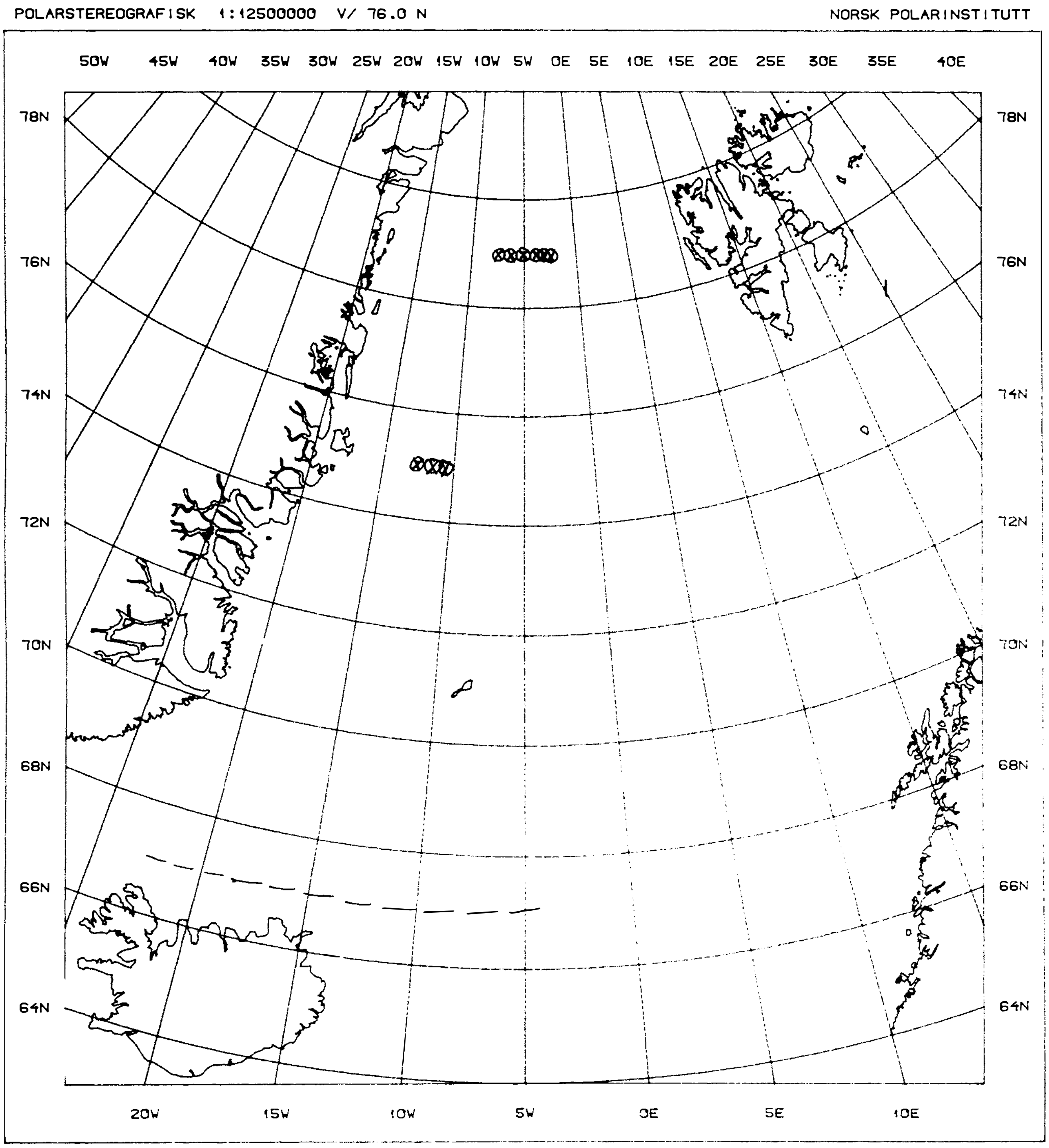
Uncertain accuracy in the displacement vectors cause errors in the mean diplacement. If UTM projection is used, controlled, but varying, errors from the projection is also introduced. The accuracy of the displacement vectors is influenced by the positional precision of the geometrical transformation of the satellite images. Good algorithms for geometrical corrections of AVHRR images, based on orbit parameters, are presented by Lauknes (1990) and Brasjö (1990). Errors introduced by insensitive frame definition by the user can also be added. This is not so important in the East Greenland Current area, but will can be more significant in an area with more diverging currents and drift directions, for example close to the polar front. If an automatic displacement program is used for the generation of the displacement vector field, it is important that a good landmask is used to avoid "sea ice drift on solid ground". A rough mask can be used as there is often fast ice close to the coast. On NOAA AVHRR images clouds must also be masked.

One important factor to be taken into consideration when NOAA AVHRR data are used for generating the displacement vector fiel, is that on cloudfree occasions the wind direction is normally from the north in the Greenland Sea. This will give a biased data set. Dech (1990) has generated displacement vector fields also on rather cloudy days and the varying directional homogenity can be seen from his displacement field maps. The use of ERS-1 data would allow independence from the weather and obtaining complementary data to study this effect. The swath width and resolution of ERS-1 is perhaps a bit too small to be optimal for sea ice drift studies with most of the Greenland Sea as an investigation area. For assessment of the general background ice displacement, about 1 km seems to be a good resolution. Tests with radar images from USSR with 1 km resolution are to be performed in the near future.

The present program combines average draft with average drift, or displacement. This gives a twodimentional variable, which can be said to represent vertical area. This is itself a good indicator of the ice fluctuations with time. However, combining the variable with some drift perpendicular parameter would give ice volume transport. The simplest way to achieve this is to use the width of the ice stream. This can be collected weekly (retrospectively) from the digital ice charts stored at the Norwegian Polar Research Institute or directly from the satellite images. The program developed by Schistad et al. (1990), for example, gives both the ice displacement and the ice edge position. Another, more accurate, method would be to use the ice concentration in equalsized units perpendicular to the drift stream as a weighting factor. Each unit could be compared digitally with the center unit (containing the ULS), thereby obtaining a percentage weighting value. This can in the long run also give a statistical material for a stream cross section. A more elaborate, but most correct, way, would be to have an array of ULSs on a cross stream line and combine the series of 2D time series to a 3D time series. The displacement must then be averaged in narrower frames, with little overlap, to keep the cross stream dynamics. ULSs were positioned along latitudes in the Greenland Sea in 1991, Fig 4.

The vertical area value can very well be a climate monitoring indicator, provided continuous measurements will take place. Some standardized time period each month or year can be stored to generate a time series. Combinations of certain time periods of the year can be used. In any time series it is very important that the parameters are extracted in the same way each time and, that the values of the whole series are comparable. Unfortunately this is definitelly not always the case. By using one and the same program, like the developed program, comparability is achieved. If any changes are made, it is very important that they are carefully recorded. One good way of doing this is to have a log included in the program, preferably in an introductory header (commented out).

The future for this type of ice measurement seems very promising. The ULS concept is very operational and several improvements have been made since the 1987 version. The newer sensors are less depth and shock sensitive (Johannesen 1990). On the new ULSs temperature and signal quality are also measured (Svein Værholm, CMI, pers. comm.). The interest for this ULS type has grown dramatically, from the one deployed in 1987, used here, to four deployed in 1990 and 16 deployed in 1991. The developed program can, maybe with some minor improvements, be a good operational tool for the generation of time series data, which can be an asset for monitoring or other scientific purposes.



Norwegian, German and USA deployments 1991

Lat 79N	2 30W 2300 m	Lat 75N	12 30W depth unknown
- " -	3 20W 2000 m	- " -	11 30W -- " --
- " -	4 00W 1500 m	- " -	10 30W -- " --
- " -	5 00W 1200 m		
- " -	6 00W 500 m	Lat 67N	Lon and depth unknown
- " -	7 00W 350 m		

Fig. 4 The deployments of ULS in the Greenland Sea in 1991.

References

Brasjö, Carina (1990): **Geometrisk korrektion av NOAA AVHRR data**. (In Swedish) (Geometric Correction of NOAA AVHRR data) Lunds Universitet, Naturgeografiska Institutionen, Seminarie uppsats Nr 18, Report 18, Yellow Series, University of Lund, Department of Physical Geography, Lund, Sweden. 90 pp.

Dech, Stefan Werner (1990): **Monitoring des Meereises in der Ostgrönlandsee im Mai 1988 mit Methoden der Fernerkundung**. (In German) (Monitoring of the Sea Ice in the East Greenland Sea May 1988 using Remote Sensing methods). DLR-Forchungsbericht 90-36 (DLR Research report 90-36) (Ph. D. thesis University of Würzburg). 280 pp.

Erlingsson, Bjørn (1991): **On the Temperature Distribution in the Ice Subsurface Water Column**. Unpublished.

Feldman, S I (????): **Make - A program for Maintaining Computer Praograms**. Bell Laboratories, Murray Hill, New Jersey. 8 pp.

Helland-Hansen, Bjørn & Nansen Fridtjof (1909): **The Norwegian Sea. It´s Physical Oceanography**. Kristiania, Det Mallingske Bogtrykkeri, Report on Norwegian Fishery and Marine Investigations Vol II., 422 pp.

Hewlett-Packard (????): **Instruktionsbok och Programmerings handledning HP-41C/41ev**. (In Swedish) (Manual and Programming Handbook HP41C/41ev)

Johannessen, Atle A (1990): **CMI ES-300 Series User´s Guide. Special applicablefor Model V**. CMI report 10124, CMI, Department of Science and Technology, Bergen, Norway. 21 pp.

Johnsen, Ånund S (1989): **Relations between Top and Bottom Ice Topography using a Scanning Sonar**, Proc. POAC 1989, University of Technology, Luleå, Sweden.

Kinsler, Lawrence E., Frey, Austin ., Coppens, Alan B & Sanders, James V. (1982): **Fundamentals of Acoustics**. 3:d ed. Wiley. 480 pp.

Lauknes, Inge (1990): **NOAA Ouick Look** (In Norwegian). Foredrag fra NOBIM-konferansen 1990 (Proc. from the NOBIM conference 1990), Report IR0409 FORUT, University of Tromsø. 118 pp.

Moritz, Richard E (1988): **Ice Budget of the Greenland Sea**. Ph. D. dissertation, Yale University. 180 pp.

Pohl, Peter., Eriksson, Gerd & Dahlquist Germund (1984): **Lärobok i numeriska metoder.** (In Swedish) (Schoolbook on Numerical Methods). 6:th ed. Liber. 261 pp.

Press, William H., Flannery, Brian P., Teukolsky, Saul A. & Vetterling William T. (1990): **Numerical Recipes in C. The Art of Scientific Computing**. Cambridge University Press. 735pp.

Schistad, Anne, Holbœk-Hanssen, Erik & Råheim Erlend (1990): **Ice Monitoring based on SAR Images, Task 1016: Test of Algorithms for the ERS-1 Application Project**. NCC-Note Bild/02/90, Norwegian Computing Center, Limited availability. 29 pp.

Sverdrup, H. U., Johnson, Martin W. & Fleming Richard H (1970): **The Oceans. Their Physics, Chemistry and general Biology**. Prentice Hall, Engelwood Cliffs, N. J. 1087 pp.

Ussisoo, Ilmar (1977): **Kartprojektioner**. (In Swedish) (Map projections) Teknisk skrift 1977/6, LMV, (Professional Paper 1977/6, National Land Survey, Sweden), Gävle.

Vinje, Torgny & Finnekåsa, Øyvind (1986): **The Ice Transport through the Fram Strait**. Norsk Polarinstitutt Skrifter Nr. 186, Norwegian Polar Research Institute, Oslo, Norway. 39 pp.

Vinje, Torgny & Berge Torstein (1989): **Upward Looking Sonar Recordings at 75N - 12W from June 1987 to June 1988**. Data report, Norsk Polarinstitutt Raportserie Nr 51, Norwegian Polar Research Institute, Oslo, Norway. 24 pp.

Vinje, Torgny (1991): **Arctic Ice Thickness Monitoring Project (AITMP). Status and Implementation**. Draft.

von Wiese, W. (1922): **Die Einwirkung das Polareises in Gronlandischen Meere auf die Nord-Atlantische Zyklonale Tätigkeit**. (In German) (The influence of the Polar Ice on the Cyclonic activity in the North Atlantic ocean). Annaler der Hydrographie 50, 1922. pp 271-280.

Zhang, Hongjiang (1988): **Greenland Sea Ice Motion determined from Satellite Imagery**. Internal report Greenland Sea Project No 7, Electromagnetics Institute, Technical University of Denmark, Lyngby, Denmark. 19 pp.

Zhang, Hongjiang (1989): **Ice Motion Fields in the Greenland Sea derived from AVHRR Imagery, MIZEX 87**. Internal report Greenland Sea Project No 30, Electromagnetics Institute, Technical University of Denmark, Lyngby, Denmark. 36 pp.

Zhang, Hongjiang (1990): **Ice Motion Tracking Algorithms**. Internal report Greenland Sea Project No 33, Electromagnetics Institute, Technical University of Denmark, Lyngby, Denmark. 46 pp.

Acknowledgements

Many people have supported me during this project and I want to mention some in particular. I'm grateful for the financial support given to the project by GRID-Arendal, Norwegian Space Center and the Norwegian Polar Research Institute (NPRI). The working facilities at NP have been excellent. Torgny Vinje, Terje Løyning, Ånund Kvambekk and Øyvind Finnekåsa, all NPRI, have given good supervision and many pieces of good advice. I'm grateful to Eberhard Fahrbach, Alfred Wegener Institute for Polar and Marine Research, Bremerhaven, Germany, for sending me temperature data from the mooring, and to Hongijang Zhang, Electromagnetics Institute, Lyngby, Denmark, for giving NP access to some of his displacement data files for the period 1987-88. Torstein Berge has given assistance in the use of the NP computers and Jostein Almlien in the use of the image processing system, IIS. Anne Schistad, Norwegian Computing Center, and Carina Brasjö, University of Lund (LU), have given advice about their programs. At the end of the project John Thingstad, University of Oslo, has given good assistance in C programming and program improvements. He also did the programming for the conversion from version 0.6 to version 1.0. Lars Kullerud at GRID-Arendal assisted on using CorelDraw for one of the figures. Torgy Vinje has kindly read and commented an early version of the report. Mary C Berge has checked the English language used. Frode Reisnes, HMS, Arendal, generously gave me access to Macintosh computers.
Susan Barr, NP, and Anders Appelquist, LU, kindly let me stay at their homes during parts of the project. I'm also grateful to all at NP and in Lund who have helped in getting a pleasant working and living environment.

Computations of the ice draft

The ice draft is computed as the difference between the depth of the sonar and the distance to the ice. Reflections from an ice-free water surface at calm weather was used for calibration of the ULS. An early calibration sequence showed a systematic error of about one dm. Inquiry to the manufacturer revealed that this represented the distance between the depth measuring transducer and the sonar transducer (Torgny Vinje, NPRI, pers. comm.). The depth transducer is located below the ultrasonic trancducer in the housing bouy of the ULS used in 1987-1988. Addition of a correction factor for this made the draft measurements much more accurate. The used correction factor is -0.13 m.

Draft = ULS_depth - Distance_to_ice_subsurface + Correction_factor [m]

The depth of the sonar (ULS_depth) is computed from the measured pressure at the bouy, reduced for the current air pressure at the time of the measurement. This gives the pressure exerted by the water. Dividing the pressure value by the pressing force of the water, density times local gravitational acceleration, gives the depth.

ULS_depth = (Measured_pressure - Air_pressure) / Water_density * Local_gravity [m]

The air pressure was recorded for 00Z and 12Z. The pressure 00Z is used for measurements between 1800 h and 0600 h and the pressure 12Z is used for measurements between 0600 h and 1800 h. If any value is missing a default value of 1012 mbar, or hPa, is used.
The density depends very much on the salinity(Nikolai Doronin, guest res. NPRI, pers. comm.). A salinity time series was unfortunatelly not available during the period. A constant salinity of 33 per mille have been used for the computations, based on CTD measurements in the area. The local gravitational acceleration for the considered position is taken to be 9.829 m/s^2.

The measured pressure is computed using calibration constants ("gain and offset") supplied by the manufacturer. A Digiquartz 8060 D. C. calibration unit is used for the calibration. The transducer is checked together with it in a pressure tank. The accuracy of the Digiquartz 8060 D. C. is said to be about one magnitude better than that of the transducer (Svein Vœrholm, CMI, pers. comm.). The pressure transducer gives a recording of a numerical unit for each measurement.

Measured_pressure = Numerical_unit * Gain_factor + Offset_factor [Pa]

Measurements with numerical units over 4095, corresponding to about 70 m below the surface was excluded. The transducer was not constructed for deeper levels, but was on occations dragged deeper. The gain and offset factors for the used transducer was 195.494 and 4369.6, respectively.

The distance to the ice subsurface is computed as one way transit time times the current soundspeed times the cosine of the tilt of the ULSbouy.

Distance_to_ice_subsurface = One_way_transit_time * Sound_speed * cos(Tilt) [m]

As the bouy turned out to be very stable the tilt values are usually small giving a cosine close to one. Measurements with a tilt exceeding 20 have been excluded. They are extremly few.

The two most similar response times, out of four, each fourth minute was stored as numerical units. The average one way transit time was computed using calibration constants ("gain and offset") supplied by the manufacturer. Half the mean of the two numerical units, representing response time, was used.

Numerical_unit_average = ((Numerical_unit_1 + Numerical_unit_2) / 2) / 2

One_way_transit_time = Numerical_unit_average * Gain_factor + Offset_factor [s]

Observations are excluded if any of the two response times exceeds 0.05 s (1952 units) or if the difference between them exceeds 0.001 s (40 units). This time difference corresponds to a draft difference of about 1.4 m (Vinje & Berge 1989). Such meassurements are considered to be involving secundary sonar echos, from some off-nadir location. The gain and offset used in the computations are 0.0000512 and 0.00004, respectively.

The sound speed is dependent on depth, water salinity and temperature. The formula used is the simplified version given by Kinsler et al. (1982).

Soundspeed = Surface_soundspeed + 4.6*T - 0.055*T^2 + 0.003*T^3 + (1.39 - 0.012*T) * (Salinity - 35) + 0.017*Depth

T = The daily mean temperature at Depth

The sea surface soundspeed used is 1449.0 m/s, which is valid for surface sea water at a salinity of 35 per mille and a temperature of 0 C. In fresh water the corresponding soundspeed is 1403 m/s.
As mentioned above salinity data has not been available. A constant value of 33 per mille is used also here. Temperature data were received during the project. The measured temperatures close to the ULS failed and the available temperature series are from 72 and 101 m below sea surface, respectively. The temperature comprise daily means. The earlier computations used a constant temperature of -1 C for the computations of soundspeed. Using a constant depth of 25 m below sealevel this gave a constant soundspeed of 1442 m/s^2. Later improved temperature assessments have been included to obtain more accurate values and to better cover the variations with time. This is described separately below.

One per mille difference in the salinity, at constant depth and temperature, gives a difference in the soundspeed of about 1.5 m/s. One degree difference in the temperature, at constant salinity and depth, gives a difference in the soundspeed of about 3 m/s. Ten meters difference in the depth, at constant temperature and salinity, gives a difference in the soundspeed of mm, or less, with the used equation, at those depths that are of interest here.

Statistics computed for the draft and ULS depth series

Statistics are computed both for the draft values and the ULS depth values. A modified version of a routine from Numerical Recipes in C (Press et al 1990) is used. The parameters computed are mean value, maximum and minimum value, distance to mean, variance, standard deviation, skewness and kurtosis. Skewness and kurtosis are computed after a check that the variance does not equal zero, which prevents the computation. The number of rejected samples is counted and not included in the computations. Unrealistic values are also excluded, for example too much negative draft or depth values too close to the surface. For the computation of draft statistics the user is asked for the limit value for exclusion of drafts. Any limit can be chosen, depending on the purpose. In the test series -0.2 m is used. When using this limit, there is some margin for inclusion of values caused by wave effects, but gross fliers are excluded. Minimum and maximum value extraction, however, uses the whole draft array, excluding only those values rejected in the draft computation process. For the ULS depth statistics a constant value of 20 is used at present.

Improvements of the temperature used for soundspeed computation

The temperature is an important variable in the computation of the soundspeed as is evident from the formula. As the series of daily temperature values wasn't available earlier, a constant temperature of -1.0 C has been used for the computation of soundspeed. The temperature series now available are from 72 and 101 meters below sealevel, respectively. Quite a variation can be seen in the temperatures.

The first, and currently implemented, step was to use the 72-m value directly. This gave at least some correction for the temperature variation. A second attempt was to use linear regression, using both values. This gave much to high temperatures, which could be expected considering the shape of the temperature curve. Using linear regression between the 72-m temperature and the seasonal ice subsurface temperature would give a better approximation of the real temperature in the water column. The temperature change with depth is, however, far from linear, as can be seen in Fig 4.

The sinus hyperbolicum function was expected to give a good fit but to use demanding computations (Bjørn Erlingsson, NPRI, pers.l comm.). The fitting of a logarithmic function was considered the optimum aproximation (Erlingsson 1991; Torgny Vinje, NPRI, pers. comm.). The logarithmic profile was locked in the upper end using the seasonal ice subsurface temperature and a zero derivative at the subsurface. Attempts to implement this have been made, but are not yet quite finished.

The logarithmic temperature profile

The logarithmic temperature equation is determined using two iterative segments, one to find a good temperature constant and kappa-value for the formula

$$\text{Temperature(Depth)} = \text{Temperature}_{SS} * e^{\text{Kappa}*(\text{Depth - Draft})} + \text{Temperature}_J * (1 - e^{\text{Kappa}*(\text{Depth - Draft})})$$

and the second to determine the most representative temperature level in the water column, here assumed to be located half way between the bouy and the ice subsurface.

The first iteration segment consists of the two formulas

$$\text{Kappa}_J = (1 / (DD - SD)) * \ln((T_{DD} - T_{J-1}) / (T_{SD} - T_{J-1}))$$

$$T_J = (T_{DD} - T_{SS} * e^{\text{Kappa}_J*(DD - \text{Draft})}) / (1 - e^{\text{Kappa}_J*(DD - \text{Draft})})$$

where T = Temperature
DD = Deep Depth (in meters or as index)
SD = Shallow Depth (in meters or as index)
SS = SubSurface (index)
J, J -1 = index of the iteration

The draft is neglected in the computations as this parameter normally is small compared to the depth of the ULS. A mean draft of 2.5 meters has also been tested to improve the approximation. If we have T_J for J = 0 such that dT / dDepth = 0 at Depth = Draft we get T_J, (J = 0) = T_{SS}. T_{SS} is season dependent. At present only two values are used, one for the period September to March and one for the rest of the year. The used values -1.75 and -1.35 respectively, are taken from (Sverdrup et al. 1970). The iteration continues until the temperature difference is below a control value or the number of iterations exceeds a certain limit.

The second iteration segment uses four steps:

Middel_depth = Depth_in / 2

$$\text{Temperature(Middle\_depth)} = \text{Temperature}_{SS} * e^{\text{Kappa(Middle\_depth)}} + \text{Temperature}_J * (1 - e^{\text{Kappa(Middle\_depth)}})$$

$$\text{Soundspeed} = \text{Surface\_soundspeed} + 4.6*T - 0.055*T^2 + 0.003*T^3 + (1.39 - 0.012*T) * (\text{Salinity} - 35) + 0.017*\text{Middel\_depth}$$

(T = Temperature)
Distance_to_ice_subsurface = One_way_transit_time * Sound_speed * cos(Tilt)

The Distance_to_ice_subsurface is used as the Depth_in in the next iteration. The first run the depth of the ULS, minus 2.5 meters, is used as Depth_in. The iteration continues until the difference in distance to ice subsurface is below a control value or the number of iterations exceeds a certain limit.

The Runge-Kutta approach

To consider the varying soundspeed with depth and temperature to a larger extent a fourth order Runge-Kutta method has been used.
Runge-Kutta methods comprise numerical methods for the solution of initial value problems. They are designed to imitate methods with Taylor series expansion, but without requiring analytic expressions for the higher order derivatives of y. They only involve evaluation of the original function f. The interval is partitioned into a finite number of subintervals. The combination of values f(x,y) from these intervals, close to the solution curve, gives good accuracy. The convergence to the exact solution is faster for higher order methods, when the spacing is decreased.

The most well-known Runge-Kutta method is the fourth order Runge-Kutta, also called the classical Runge-Kutta. It employs the four following strategic equations:

$k_1 = h ¤ f(x,y)$
$k_2 = h ¤ f(x_n + h/2, y_n + k_1/2)$
$k_3 = h ¤ f(x_n + h/2, y_n + k_2/2)$
$k_4 = h ¤ f(x_n + h, y_n + k_3)$

$y_{n+1} = y_n + 1/6 ¤ (k_1 + 2¤k_2 + 2¤k_3 + k_4)$

h = subinterval, step.

The error is $O(h^4)$.
A very simple example to illustrate the method is the solution of the differential equation:

$y' = x + y$
$y(0) = 1$

taken from Pohl et al. (1984).

h = 0.1

x	y	f=x+y	k=h¤f
0	1	1	0.1
0.05	1.05	1.1	0.11
0.05	1.055	1.105	0.1105
0.1	1.1105	1.2105	0.12105

$^1/_6$ ¤ (0.1 + 2¤0.11 + 2¤0.1105 + 0.12105) = 0.110342

0.1	1.110342	1.210342	0.121034
0.15	1.170859	1.320859	0.132086
0.15	1.176385	1.326385	0.132638
0.2	1.242980	1.442980	0.144298

$^1/_6$ ¤ (0.121034 + 2¤0.132086 + 2¤0.132638 + 0.144298) = 0.132463

y(0.2,0.1) = 1.110342 + 0.132463 = 1.242805

The correction term equals 0, i. e. the error is negligible and all six decimals are significant.

The method is implemented in the program by John Thingstad.
Starting from the depth of the ULS buoy the distance to the ice subsurface is computed by integration of a number of subdistances. Each subdistance is the vertical component of the sound path passed during one timestep. It is computed using the temperature representative for that depth.

The temperature as a function of depth is found by solving:

$y_1 = a \text{ ¤ } e^{bx_1} + c$

$y_2 = a \text{ ¤ } e^{bx_2} + c$ for a, b and c,

$y_3 = a \text{ ¤ } e^{bx_3} + c$ x = depth y = temperature

i e:

$b = (x_3 - x_1)/(x_1 \text{ ¤ } (x_2 - x_3)) \text{ ¤ } \ln((y_3 - y_1)/(y_2 - y_1))$

$a = (y_2 - y_1)/(e^{bx_2} - e^{bx_1})$

$c = y_1 - a \text{ ¤ } e^{bx_1}$

This is possible using the three temperature point measurements, at deep depth, shallow depth and the ice subsurface.
The current depth is then inserted to get the temperature at that depth.

The soundspeed corresponding to a subdistance is computed using fourth order Runge-Kutta and the formula from (Kinsler et al, 1982).

k_1 = soundspeed(temp(depth), salinity, depth)
k_2 = soundspeed(temp(depth + step/$_2$ ¤ k_1), salinity, depth + step/$_2$ ¤ k_1)
k_3 = soundspeed(temp(depth + step/$_2$ ¤ k_2), salinity, depth + step/$_2$ ¤ k_2)
k_4 = soundspeed(temp(depth + step ¤ k_3), salinity, depth + step ¤ k_3)

Salinity is held constant at present.
The step used is transit time /$_{10}$.

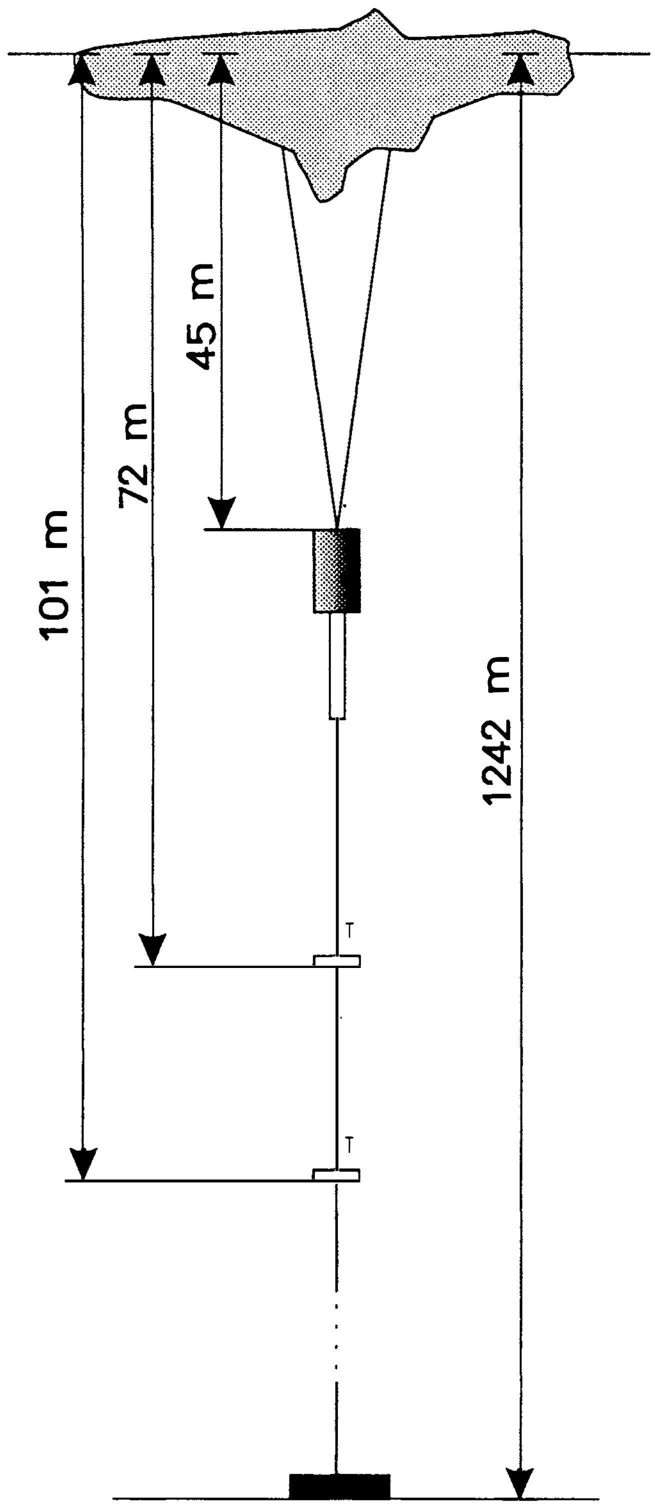
current_soundspeed = (k_1 + 2¤k_2 + 2¤k_3 + k_4)/$_6$

current_distance = step ¤ current_soundspeed

The distance is translated to vertical distance and compensated for refraction with the use of angle and speed from the previous step.

vertical_subdistance = current_distance ¤
cos(current_soundspeed / previous_soundspeed ¤ previous_angle

Finally draft = depth - Σ subdistances.



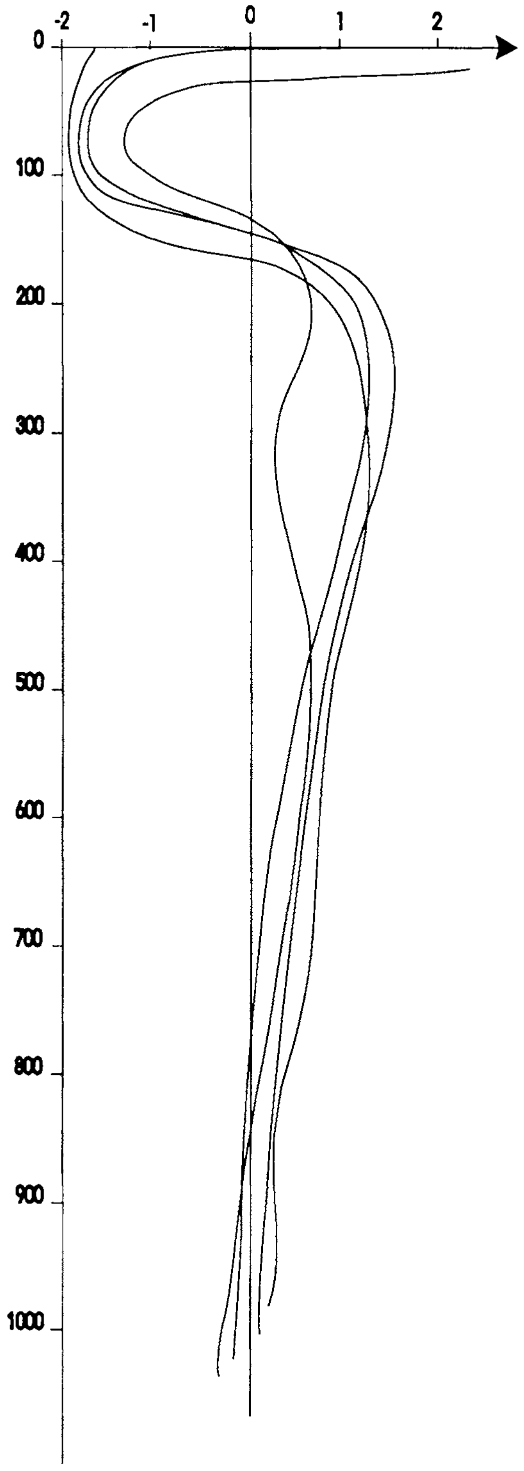


Fig. 5 The mooring configuration

Fig. 6 Temperature profiles in the EGC. After (Helland-Hansen & Nansen 1909)

The algorithm for the draft unit is roughly as follows:

1. Ask the user for the desired time interval (start time and end time). Year, month and day are compulsory. Hour and minute are optional. Read as character strings.

2. Convert the given time values from character to integer format. Separate different time parameters, hour, year etc. Compute Julian daynumber and minute-number in day.

3. Combine the search-string using year (minus century), julian day and minute number in day.

4. Compute the number of days in the given time interval. Compute the probable number of samples within the time interval. Compute the offset value for the data file (in bytes) to the first sample of the interval.

5. Combine path, file prefix, year, month and file type to one unit for the first ULS data file (month) of the interval. Try to open the corresponding file for reading. If not succeeded or if the file doesn't exist on the directory, give error message and exit.

6. If successfully opened, move to the position defined by the computed offset. Finish that line and check if the following one is the start record. If not, read out the line and check if next line is the desired start record. Continue until the record time is equal to or greater than the desired starting time. Usually this means passing one or two records.

7. When the start record is found, read the ULS values from the file and check if they are within reasonable limits. If not, set mark and continue to next record. In reading the rest of the data line check if the data are followed by any of the specified marking characters. If so, store a value in a "marking" array and add one to the (appropriate) counter.

8. If the values are considered correct compute the water pressure and transit time.

9. On the first run, read all air pressures from the air pressure file and store in an array. Each day will have two values, one for 00Z and one for 12Z.
For the following computations locate the current time in the air pressure array. Read the air pressure for the twelve hour interval that encompasses the current time.

10. Find the depth of the ULS by using the water pressure, adjusted for the contemporary air pressure, salinity and local gravitational force.

11. Interpolate the temperature half way between the ULS and the ice subsurface. Use the temperature to compute a probable soundspeed for the occasion.
Version 1.0 uses the local temperature and soundspeed at each subdistance.

12. Find the distance to the ice subsurface using the transit time and the computed sound speed.

13. Compute the draft as the difference between the depth of the ULS and the distance to the ice subsurface.

14. Store time, draft and ULS depth in separate arrays.

15. Repeat 7-14 for as many records as the computed number of samples minus a few. If end of file is encountered change to the following ULS data file (if it exists) and continue.

16. Check the few remaining records to find the finishing record of the interval, repeating 7-14 for the computation.

17. When the whole interval is past, or no more ULS data file is found, compute statistical parameters for draft and ULS depth data. Mean, minimum and maximum, deviation from mean, standard deviation, variance, skewness and kurtosis are to be computed. Skewness and kurtosis can't be computed if the variance equals zero. Values below a certain value, given by the user, are not included. This is to exclude values of to much negative draft or depth values to close to the surface.

18. If the user requests it, print the statistical values on a file also including position, interval, temperature(s) and sound speed.

19. If the user requests it, tabulate the data arrays on a file in ASCII format. Values exept draft is optional. Header is optional.

Drift computations - weighted average

The drift computations make use of user defined frames, one to five in the present version. If any end of a displacement vector falls within a frame it is given the weight associated with that frame. Weights are given by the user. Rectangular or circular frames are optional.

If rectangular frames are chosen the program uses a UTM coordinate system, centered at a user-given meridian. The corner coordinates for each frame is computed as a plus/minus offset from the centerpoint.

Frame_corner = (Center_x +/- User_given_x_offset , Center_y +/- User_given_y_offset)

Any vector end coordinates are transformed to UTM and matched with the corner coordinates. If the x coordinate falls between the border lines in the x direction, y is checked similarly, otherwise not. If it is to be included in the computations, it's x- and y-component is computed as the difference in UTM values.

Delta_y = $y_2 - y_1$ Delta_x = $x_2 - x_1$ (in meters)

The component values are then multiplied with the frame weight and summed. The resultant components are computed as the weighted average, i e :

X_resultant_component = Σ(Weight*Delta_x) / Σ(Weight)

Y_resultant_component = Σ(Weight*Delta_y) / Σ(Weight)

The length of the resultant is computed using the formula of Pytagoras:

Resultant_length = Sqrt(X_resultant_component^2 + Y_resultant_component^2) [m]

The azimut is computed as :

Azimut = Arctan(X_resultant_component / Y_resultant_component) + 180

If circular frames are chosen the program uses the global coordinates, latitude and longitude, directly. The computations are made in radians. The radius vector for each end point of a displacement vector is computed in km using the formula below, taken from the HP manual.

If Lat_1 = latitude of vector start point
    Lon_1 = longitude of vector start point
    Lat_2 = latitude of center point
    Lon_2 = longitude of center point
    Lat_3 = latitude of vector end point
    Lon_3 = longitude of vector end point

all in degrees, then

Temp_start = Sin(Lat_1)*Sin(Lat_2) + Cos(Lat_1)*Cos(Lat_2)*Cos(Lon_1 - Lon_2)
Temp_end = Sin(Lat_3)*Sin(Lat_2) + Cos(Lat_3)*Cos(Lat_2)*Cos(Lon_3 - Lon_2)

and finally

Radius_start = 60.00*Arccos(Temp_start)*Convertion_factor _to_km
Radius_end = 60.00*Arccos(Temp_end)*Convertion_factor _to_km

The convertion factor used is from nautical miles to km and by international definition 1.852.

If any of the vector radii are shorter than the radii of a given frame it is given the weight of that frame. If the end points of a vector are located in different frames the vector gets the weight of the centermost frame. For any vector that is included the meridional and perpendicular component is computed as the difference in latitude and longitude, respectively. The mean latitude and longitude are also computed and the later used to obtain the center of gravity in the weighted displacement vector field.

Delta_phi = Polemost_phi - Equatormost_phi
Delta_lambda = GM_remote_lambda - GM_close_lambda

Mean_latitude = Delta_phi / 2 + Equatormost_phi
Mean_longitude = Delta_lambda / 2 + GM_close_lambda

Delta_lambda will be negative west of the Greenwich meridian, GM, if the meridians are enumerated negatively to the west. If 360 longitude system is used the formula for Delta_lambda will be reversed. Absolute values are used.

Delta_lambda is normalized using Cos(Mean_latitude). This is valid for a spherical earth as the length of an arc segmant, delta_lambda, at some latitude is equal to delta_lambda*Earth_radius*Cos(that latitude). This is illustrated in Fig. 5.

The component values are then multilpied with the frame weight and summed. The mean latitude and longitude are also summed.
The resultant components are computed as the weighted average, i. e. :

Phi_resultant_component = Σ(Weight*Delta_phi) / **S**(Weight)
Lambda_resultant_component = Σ(Weight*Delta_lambda * Cos(Mean_latitude) /
Cos(Tie_latitude) * Σ(Weight)

where

Tie_latitude = Σ(Weight*Mean_latitude) / ΣWeight

Tie_longitude = Σ(Weight *Mean_longitude) / ΣWeight

The resultant in radians is computed using spherical trigonometry, assuming perpendicular components. The components are also assumed to be in the meridional and perpendicular direction. This gives a right angle between them and simplifies the computations.

Resultant_in_radians = Arccos(Cos(Phi_resultant_component)*
Cos(Lambda_resultant_component))

Angle_in_radians = Arccos((Cos(Mean_longitude) - Cos(Resultant_in_radians)*
Cos(Mean_latitude)) / (Sin(Resultant_in_radians) * Sin(Mean_latitude)))

The angle is to the left of the right angle in the two quadrants that have equal axis signs and to the right in the two quadrants having opposite axis signs.
The resultant is converted to km using the radius corresponding to the mean curvature (Ussisoo 1977) at the point representing the center of gravity of the displacement vector field.

Mean_curvature = Half_minor_axis*Sqrt(1- Excentricity^2) /
(1 - Excentricity^2*Sin^2(Tie_latitude))
Resultant_length = Resultant_in_radians*Mean_curvature

Half minor axis and excentricity above refers to the earth ellipsoid. Used values are WGS84, 6378137 m and 0.00669438, respectively.

The azimut is the angle in radians converted to degrees and added 180 (if directed south).

Standard deviation is computed for each component both in the rectangular case and the circular case.

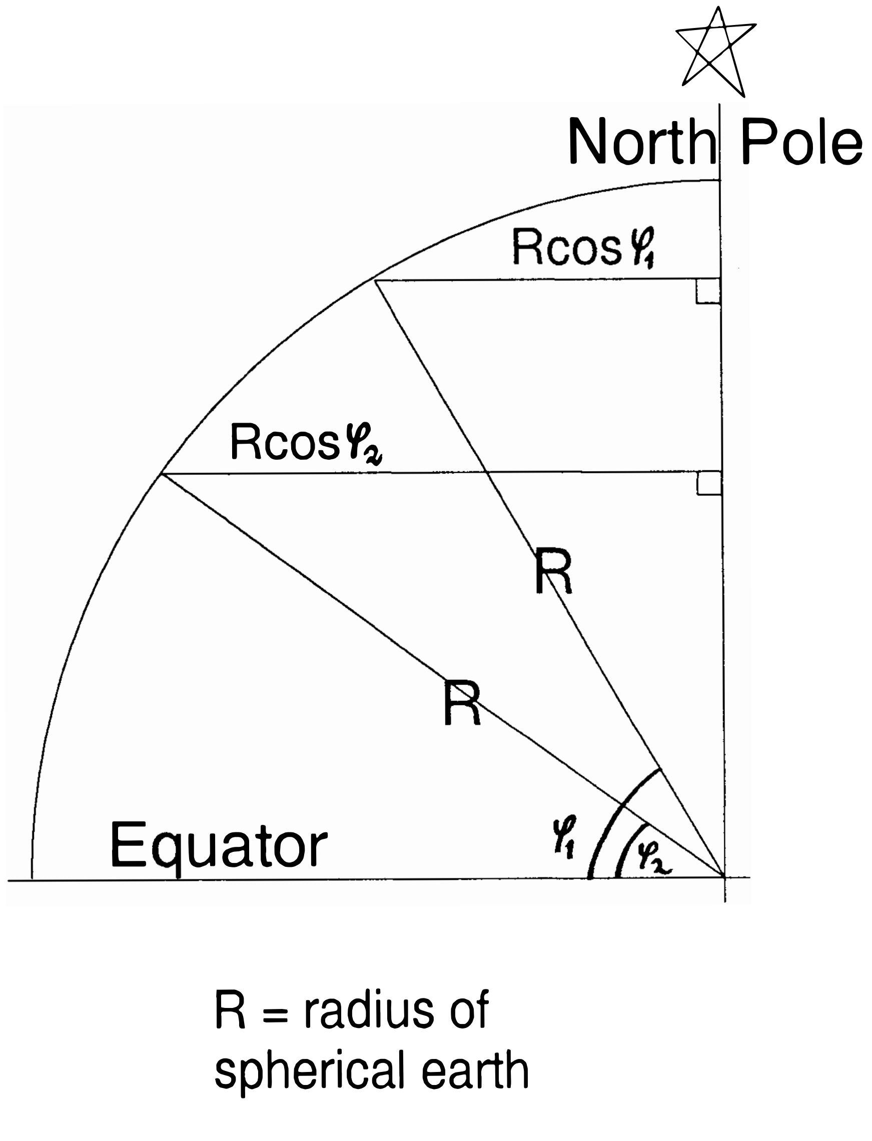


Fig. 7 The perpendicular radius for a spherical Earth.

The algorithm for the drift averaging unit is roughly as follows:

1. Ask the user for the type of weighing frames, the number of frames, their dimensions and weights.

2. If rectangular frames are chosen transform to UTM coordinate system. Find the frame corners in UTM coordinates.

3. Get the file format from the user, open the file with the drift vector coordinates in latitude and longitude.

4. Read past some header lines

5. Read the data records sequentially.

6. If rectangular frames are chosen transform the vector coordinates to UTM. Check if any end of the vector falls within any of the frames. If circular frames, compute the distance from the vector end points to the center point. Check if the distance from any of the end points of the vector lies within any of the frames. **If** one of the end points of the displacement vector falls **within** a frame, give it the weight of that frame. If it falls within two, give it the weight of the centermost frame.

7. Add vector components to sums and squared sums. For rectangular frames the components are delta x and delta y, in UTM coordinates and for circular frames the components are delta phi (latitude) and delta lambda (longitude) times phi. The longitudinal component is multiplied by phi for normalization.

8. Repeat 5-7 until end of file is reached.

9. Use the computed sums to compute mean displacement, azimut and component standard deviation. In the rectangular case, plane trigonometry is used and in the circular case, spherical trigonometry is used. The local mean curvature is used for spherical computation.

Subroutine overview and call structure, version 0.6

```
Main
 1        ulsadm
 1        1        imatrix (irutol.h)
 1        1        ivector (irutil.h)
 1        1        vector (nrutil.h)
 1        1        free_imatrix (irutil.h)
 1        1        free_vector (nrutil.h)
 1        time_in
 1        ice_conv
 1        1        leap
 1        get_temp_current
 1        1        leap
 1        1        read_rest
 1        find_compute
 1        1        make_ulsfile
 1        1        draft_comp
 1        1        1        get_air_pressure
 1        1        1        1        read_rest
 1        1        get_next_file
 1        1        1        make_ulsfile
 1        1        1        1        ead_rest
 1        1        momentx
 1        1        1        nrerror (irutil.h)
 1        1        print_on_file
 1        1        print_table_file
 1        (end ulsadm)
 1        driftadm
 1        1        sum_init
 1        1        get_drift_file
 1        1        ask_frame
 1        1        frame_in
 1        1        set_rect_corners
 1        1        1        gtu (utm.h)
 1        1        read_first_on_file
 1        1        1        read_rest
 1        1        read_coor
 1        1        1        read_rest
 1        1        sort_vect_out_sum
 1        1        1        sum_dl_dp
 1        1        1        sum_dx_dy
 1        1        1        gtu (utm.h)
 1        1        get_mean_lp
 1        1        1        mean_curvature
 1        1        get_mean_xy
 1        (end driftadm)
(end main)
```

Subroutine overview and call structure, version 1.0

```
main
1       draft
1       1       promt_date
1       1       1       frontstrip
1       1       1       validate
1       1       1       normalized_year
1       1       promt_time
1       1       1       frontstrip
1       1       1       validate
1       1       date_time_to_minutes
1       1       1       date_to_days_since_1980
1       1       1       1       leap_year
1       1       1       1       date_to_day_nr
1       1       1       1       1       leap_year
1       1       1       time_to_minutes
1       1       promt_real
1       1       1       frontstrip
1       1       1       validate
1       1       init_ULS_file
1       1       1       promt_file
1       1       1       1       frontstrip
1       1       1       1       validate
1       1       1       ULS_file_name
1       1       1       1       minutes_to_v_time
1       1       1       1       1       days_to_years
1       1       1       1       1       1       leap_year
1       1       1       1       day_nr_to_date
1       1       1       1       1       leap_year
1       1       1       find_ULS_data
1       1       1       1       minutes_to_v_time
1       1       1       1       1       days_to_years
1       1       1       1       1       1       leap_year
1       1       1       1       find_ULS_pack
1       1       1       read_ULS_data
1       1       1       1       read_ULS_pack
1       1       1       1       extract_ULS_time
1       1       1       1       normalized_year
1       1       1       1       v_time_to_minutes
1       1       1       1       1       leap_year
1       1       1       1       ULS_pack_to_data
1       1       init_S_TEMP_file
1       1       1       promt_file
1       1       1       1       frontstrip
1       1       1       1       validate
1       1       1       init_TEMP_header
1       1       1       1       read_TEMP_data_header
```

```
1       1       1       1       1       read_TEMP_pack_header
1       1       1       1       1       1       (strtod)
1       1       1       1       1       date_time_to_minutes
1       1       1       1       1             date_to_days_since_1980
1       1       1       1       1       1       1         leap_year
1       1       1       1       1       1       1         date_to_day_nr
1       1       1       1       1       1       1       1     leap_year
1       1       1       1       1       1       1        time_to_minutes
1       1       1       1       1       TEMP_pack_to_data
1       1       1       find_S_TEMP_data
1       1       1       1       time_to_file_day
1       1       1       1       1       day_difference
1       1       1       1       1       1       minutes_to_days
1       1       1       1       find_S_TEMP_pack
1       1       1       read_S_TEMP_data
1       1       1       1       read_S_TEMP_pack
1       1       1       1       file_day_to_time
1       1       1       1       S_TEMP_pack_to_date
1       1       init_D_TEMP_file
1       1       1       promt_file
1       1       1       1       frontstrip
1       1       1       1       validate
1       1       1       init_TEMP_header
1       1       1       1       read_TEMP_data_header
1       1       1       1       1       read_TEMP_pack_header
1       1       1       1       1       1       (strtod)
1       1       1       1       1       date_time_to_minutes
1       1       1       1       1       1     date_to_days_since_1980
1       1       1       1       1       1       1         leap_year
1       1       1       1       1       1       1         date_to_day_nr
1       1       1       1       1       1       1       1     leap_year
1       1       1       1       1       1       1        time_to_minutes
1       1       1       1       1       TEMP_pack_to_data
1       1       1       find_D_TEMP_data
1       1       1       1       time_to_file_day
1       1       1       1       1       day_difference
1       1       1       1       1       1       minutes_to_days
1       1       1       1       find_D_TEMP_pack
1       1       1       read_D_TEMP_data
1       1       1       1       read_D_TEMP_pack
1       1       1       1       file_day_to_time
1       1       1       1       D_TEMP_pack_to_data
1       1       init_AIR_PR_data
1       1       1       promt_file
1       1       1       1       frontstrip
1       1       1       1       validate
1       1       1       find_AIR_PR_store
1       1       1       1       minutes_to_v_time
```

```
1       1       1       1       1       days_to_years
1       1       1       1       1       1       leap_year
1       1       1       read_AIR_PR_store
1       1       1       1       read_AIR_PR_pack
1       1       1       1       extract_AIR_PR_time
1       1       1       1       1       normalized_year
1       1       1       1       v_time_to_minutes
1       1       1       1       1       leap_year
1       1       1       1       AIR_PR_pack_to_store
1       1       get_S_TEMP_header
1       1       get_D_TEMP_header
1       1       init_DRAFT_file
1       1       1       promt_bool
1       1       1       promt_file
1       1       1       1       frontstrip
1       1       1       1       validate
1       1       1       promt_draft_info
1       1       1       1       frontstrip
1       1       1       1       validate
1       1       1       write_draft_header
1       1       init_STAT
1       1       1       (tmpfile)
1       1       init_STAT_file
1       1       1       promt_bool
1       1       1       promt_file
1       1       1       1       frontstrip
1       1       1       1       validate
1       1       get_ULS_data
1       1       1       copy_ULS_data
1       1       1       read_ULS_data
1       1       1       init_ULS_file
1       1       get_S_TEMP_data
1       1       1       copy_S_TEMP_data
1       1       1       read_S_TEMP_data
1       1       get_AIR_PR_data
1       1       1       copy_AIR_PR_data
1       1       1       read_AIR_PR_data
1       1       1       extract_AIR_PR_data
1       1       1       1       minutes_to_v_time
1       1       1       1       1       days_to_years
1       1       1       1       1       1       leap_year
1       1       1       1       minutes_to_time
1       1       get_SUR_TEMP_data
1       1       1       minutes_to_v_time
1       1       1       1       days_to_years
1       1       1       day_nr_to_date
1       1       1       1       leap_year
1       1       1       date_to_day_nr
```

```
1       1       1       1       leap_year
1       1       1       v_time_to_minutes
1       1       1       1       leap_year
1       1       transit_time
1       1       sonar_depth
1       1       1       sensor_depth
1       1       1       1       water_pressure
1       1       1       1       pas
1       1       compute_ice_draft
1       1       1       compute_EXP_const
1       1       1       rad
1       1       1       distance
1       1       1       1       rel_speed
1       1       1       1       1       water_sound_speed
1       1       1       1       1       1       temp
1       1       1       1       1       1       1       exp_func
1       1       update_STAT
1       1       put_DRAFT_data
1       1       1       minutes_to_v_time
1       1       1       1       days_to_years
1       1       get_STAT
1       1       put_STAT_data
1       1       1       minutes_to_v_time
1       1       1       1       days_to_years
1       1       1       day_nr_to_date
1       1       1       1       leap_year
1       1       1       minutes_to_time
1       (end draft)
1       drift
1       1       suminit
1       1       get_drift_file
1       1       1       promt_file
1       1       1       1       frontstrip
1       1       1       1       validate
1       1       1       promt_integer
1       1       1       1       frontstrip
1       1       1       1       validate
1       1       ask_frame
1       1       1       promt_integer
1       1       1       1       frontstrip
1       1       1       1       validate
1       1       frame_in
1       1       1       promt_real
1       1       1       1       frontstrip
1       1       1       1       validate
1       1       set_rect_corners
1       1       read_first_on_file
1       1       read_coor
```

```
1       1       sort_vect_out_sum
1       1       1       sum_dl_dp
1       1       1       gtu
1       1       1       sum_dx_dy
1       1       get_mean_lp
1       1       1       mean_curvature
1       1       get_mean_xy
1       (end drift)
(end main draftdrift)
```

```
#
# makefile for drafdrif.exe
# Made in Turbo C++ version 1.0 on IBM pc
#

.c.obj:
    @tcc -ml -O -Z -a -c -Ic:\tc\include -Lc:\tc\lib { $< }

#link file

drafdrif.exe: validate.obj datetime.obj prompt.obj draftout.obj \
        temp.obj airpr.obj uls.obj draft.obj drift.obj drafdrif.obj

        @tlink /c  @response.txt,drafdrif,, c:\tc\lib\c0s.lib \
          c:\tc\lib\emu.lib


#compile files

validate.obj: validate.c validate.h
datetime.obj: datetime.c datetime.h \
        validate.h validtab.h
prompt.obj:   prompt.c   prompt.h \
        globals.h validate.h
draftout.obj: draftout.c draftout.h \
        globals.h validate.h datetime.h prompt.h
temp.obj:     temp.c  temp.h \
        globals.h datetime.h
airpr.obj:    airpr.c airpr.h \
        globals.h datetime.h
uls.obj:      uls.c uls.h \
        globals.h datetime.h

draft.obj:    draft.c draft.h \
        globals.h datetime.h \
        uls.h temp.h airpr.h prompt.h draftout.h

drift.obj:    drift.c drift.h \
        globals.h prompt.h

drafdrif.obj: drafdrif.c draft.h drift.h
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
/*globals.h*/
#ifndef GLOBALS
#define GLOBALS

#define PRIVATE static
#define PUBLIC

typedef enum {false, true} bool;

#endif
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
/*validate.h*/
/*
 * prototype for validate.cpp
 *
 * formated string parser
 */
```

```
#ifndef GLOBALS
#include "globals.h"
#endif

bool PUBLIC validate(const char *, const char *);
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
/*validtab.h*/
/*
 * table of legal commands
 *
 * used in validate.cpp and chars.cpp
 */

#define S_printable    '.'  /* printable character                    */
#define S_digit        '#'  /* digit                                  */
#define S_letter       'A'  /* letter                                 */
#define S_upper_case   'U'  /* uppercase letter                       */
#define S_lower_case   'L'  /* lowercase letter                       */
#define S_space        'S'  /* skip space                             */
#define S_ignore_case  '~'  /* ignore case of preseding character in input*/
#define S_litteral     '/'  /* predesessor characters litteral value  */

#define S_1_or_more    '+'  /* 1 or more occurences of pred. item in str. */
#define S_1_or_0       '?'  /* either none or a occurence of pred.
                                  item in str.                        */
#define S_0_or_more    '*'  /* 0 or more occurences of pred. item in str. */

#define S_or           '|'  /* (a | b | c) choose a or b or c         */

#define S_begin        '('  /* begin list                             */
#define S_end          ')'  /* end list                               */
#define S_and          ','  /* separate commands                      */
#define S_term         '\0' /* end of string                          */
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
/*validate.c*/
#include <stdlib.h>
#include <stdio.h>
#include <conio.h>
#include <ctype.h>
#include <string.h>
#include "globals.h"
#include "validtab.h"


/* #define DEBUG to activate modute auto integrity check */
/* #define DEBUG */
/* #define TEST here to make module executable */
/* #define TEST */


/*
 * prototypes
 */
#ifdef DEBUG
void PRIVATE Rin(char *);
void PRIVATE Rout(char *);
void PRIVATE Sin(char *name, const char **format,
                 const char **string);
void PRIVATE Sout(char *name, const char **format,
                              const char **string);
#endif

void PRIVATE error(void);

bool PRIVATE command(const char **, const char **);
bool PRIVATE parentesis(const char **, const char **);
bool PRIVATE operators(const char **, const char **);
bool PRIVATE and_sequence(const char **, const char **);
bool PRIVATE or_sequence(const char **, const char **);

bool PUBLIC validate(const char *, const char *);

/*
 * string sequence : function definitions
 *
 */

/* get next sequence */
const char *next(const char *str)
{
  if (*str != '\0')
  {
    str++;
    while (isspace(*str))
      str++;
  }

  return str;
}

/* get next character */
const char *nextch(const char *str)
{
  if (*str != '\0')
    str++;

  return str;
}

int get(const char *str)
{
  return (int) *str;
}

const char *skip(const char *str)
{
  while (isspace(*str))
    str++;

  return --str;
}

#ifdef DEBUG
FILE PRIVATE *file_log = NULL;
FILE PRIVATE *R_log = NULL;
#endif

/*
 * routines used strictly for debugging
 *
 * When debugging a recursive decent compiler/interpeter we wish to trace
 * which routine are entered.
 *
 * Rin(char *name)
 *   When entering procedure print "Entering: <function name>"
 *
 * Rout(char *name)
 *   When leaving procedure print "Leaving  : <function name>"
 *
 * auto indents text with 5 spaces for each recursion level
 * (for enchanced readabillity)
 */

#ifdef DEBUG
int PRIVATE level;  /* initially set to 0 */

void PRIVATE Rin(char *name)
{
  int i;

  fputs("Entering : ", R_log);
  for (i = 0; i < level; i++)
    fputs("     ", R_log);
  fprintf(R_log, "%s\n", name);

  level++;
}

void PRIVATE Rout(char *name)
{
  int i;

  level--;

  fputs("Leaving  : ", R_log);
  for (i = 0; i < level; i++)
    fputs("     ", R_log);
  fprintf(R_log, "%s\n", name);
}

void PRIVATE Sin(char *name, const char **format,
                        const char **string)
{
  fprintf(file_log, "ENTERING: %s\n", name);
  fprintf(file_log, "          %s\n", *format);
  fprintf(file_log, "          %s\n", *string);
}

void PRIVATE Sout(char *name, const char **format,
                        const char **string)
{
  fprintf(file_log, "LEAVING:  %s\n", name);
  fprintf(file_log, "          %s\n", *format);
  fprintf(file_log, "          %s\n", *string);
}
#endif

/*
 * function:      error
 * purpose:       handle error in format string
 * Arguments:     none
 */
void PRIVATE error(void)
{
 fprintf(stderr, "\nError in format\n");
}


/*
 * function:        command
 *
 *   accepts:
 *
 *   S_printable
 *   S_digit
 *   S_letter
 *   S_upper_case
 *   S_lower_case
 *   S_ignore_case
 *   S_litteral
 */

#pragma argsused
bool PRIVATE command(const char **format, const char **string)
{
 bool val = true;

#ifdef DEBUG
 Rin("command");
 Sin("command", format, string);
#endif

 switch(get(*format))
 {
  case S_printable:
        val = (isprint(get(*string)) ? true : false);
        break;
  case S_digit:
        val =  (isdigit(get(*string)) ? true : false);
        break;
  case S_letter:
        val =  (isalpha(get(*string)) ? true : false);
        break;
  case S_upper_case:
        val =  (isupper(get(*string)) ? true : false);
        break;
  case S_lower_case:
        val =  (islower(get(*string)) ? true : false);
        break;
  case S_space:
        val = (isspace(get(*string)) ? true : false);
        *string = nextch(*string);
        *string = skip(*string);
        break;
  case S_ignore_case:
        *format = nextch(*format);
        val = ((toupper(get(*format))  == toupper(get(*string)))
             ? true : false);
        break;
  case S_litteral:
        *format = nextch(*format);
        val =  ((get(*format) == get(*string)) ? true : false);
```

```
          break;

    /* reserved symbols illegal */
    case S_1_or_0:
    case S_1_or_more:
    case S_0_or_more:
    case S_begin:
    case S_end:
    case S_term:
          error();

    default:
          if (isprint(get(*format)))
            val =  ((get(*format) == get(*string)) ? true : false);
          else
            error();
  }

  *format = next(*format);
  *string = nextch(*string);

#ifdef DEBUG
  Rout("command");
  Sout("command", format, string);
#endif

  return val;
}


bool PRIVATE parentesis(const char **format, const char **string)
{
  bool val =  true;

#ifdef DEBUG
  Rin("parentesis");
  Sin("parentesis", format, string);
#endif

  if (get(*format) == S_begin)
  {
    *format = next(*format);
    val = or_sequence(format, string);

    if (get(*format) != S_end)
      error();

    *format = next(*format);
  }
  else
    val = command(format, string);

#ifdef DEBUG
  Rout("parentesis");
  Sout("parentesis", format, string);
#endif

  return val;
}
```

```
bool PRIVATE operators(const char **format, const char **string)
{
 const char *format_pos = *format;
 const char *string_pos;

 bool val = true;

#ifdef DEBUG
 Rin("operators");
 Sin("operators", format, string);
#endif


 switch(get(*format))
 {
  case S_1_or_0:
        *format = next(*format);
        string_pos = *string;
        if(parentesis(format, string) == false)
          *string = string_pos;
        val = true;
        break;
  case S_1_or_more:
        *format = next(*format);
        val = parentesis(format, string);
        if (val == false) break;
  case S_0_or_more:
        do
        {
          *format = next(format_pos);
          string_pos = *string;
          val = parentesis(format, string);
        } while (val == true);
        *string = string_pos;
        val = true;
        break;
  default:
        val = parentesis(format, string);
 }

#ifdef DEBUG
 Rout("operators");
 Sout("operators", format, string);
#endif

 return val;
}


bool PRIVATE and_sequence(const char **format,
                           const char **string)
{
 bool val = true;

#ifdef DEBUG
 Rin("and_sequence");
 Sin("and_sequence", format, string);
#endif

 val = operators(format, string);
```

```
  while (get(*format) == S_and)
  {
    *format = next(*format);
    val = (operators(format, string) && val ? true : false);
  }

#ifdef DEBUG
  Rout("and_sequence");
  Sout("and_sequence", format, string);
#endif

  return val;
}

bool PRIVATE or_sequence(const char **format,
                         const char **string)
{
  const char *prev     = *string;
  const char *accepted = *string;

  bool val = false;

#ifdef DEBUG
  Rin("or_sequence");
  Sin("or_sequence", format, string);
#endif

  if (and_sequence(format, string) == true)
  {
    val = true;
    accepted = *string;
  }

  while (get(*format) == S_or)
  {
    *format = next(*format);
    *string = prev;
    if (and_sequence(format, string) == true)
    {
      val = true;
      accepted = *string;
    }
  }

  *string = accepted;

#ifdef DEBUG
  Rout("or_sequence");
  Sout("or_sequence", format, string);
#endif

  return val;
}


bool PUBLIC validate(const char *format, const char *string)
{
  bool val;
  const char *form = format;
  const char *str = string;
```

```
#ifdef DEBUG
  if ((R_log = fopen("Rlog.io", "w")) == NULL)
  {
    perror("Error opening file Rlog");
    return 1;
  }

  if ((file_log = fopen("log.io", "w")) == NULL)
  {
    perror("Error opening file log");
    return 1;
  }

  Rin("validate");
  Sin("validate",  &form,  &str);
#endif

   val = or_sequence(&form, &str);
   if (get(form) != S_term)
     error();
   val = (((strlen(str) == 0) || (str[0] == '\n')) && val ? true : false);

#ifdef DEBUG
  Rout("validate");
  Sout("validate",  &form,  &str);

  fclose(R_log);
  fclose(file_log);
#endif

   return val;
}

#ifdef TEST
int main()
{
  char format[256];
  char string[256];

  char *p = format;

  do
  {
    printf("\nEnter a format : ");
    fgets(format, 255, stdin);
    if (strlen(format) == 1) break;

    do
    {
      printf("\nEnter a string : ");
      fgets(string, 255, stdin);
      if (strlen(string) == 1) break;

      while (*p != '\n')
              p++;
      *p = '\0';

      p = string;
      while (*p != '\n')
              p++;
      *p = '\0';
    if (validate(format, string))
          puts("string accepted");
    else
          puts("string not accepted");

#ifdef DEBUG
    fclose(file_log);
    fclose(R_log);
#endif
   } while (true);
  } while(true);

  return 0;
}
#endif
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
/*datetime.h*/
/*
 * header file for datetime.c
 */

#ifndef DATETIME
#define DATETIME

typedef unsigned long i_time;

struct u_date
{
  int day;
  int month;
  int year;
};

struct u_time
{
  int hour;
  int min;
};

struct v_time
{
  int year;
  int day_nr;
  int min_nr;
};


int time_to_minutes(const struct u_time *);
struct u_time minutes_to_time(int);
int normalized_year(int);
int leap_year(int);
int date_to_day_nr(const struct u_date *);
struct u_date day_nr_to_date(int, int);
unsigned long date_to_days_since_1980(struct u_date *);
i_time date_time_to_minutes(struct u_date *, struct u_time *);
void minutes_to_days(i_time , int *days, int *);
void days_to_years(int , int *, int *);
struct v_time minutes_to_v_time(i_time);
i_time v_time_to_minutes(struct v_time *);
```

```
int day_difference(i_time, i_time);

#endif
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
/*datetime.c*/
/*-------------------------------------------------------------------------
 * Module:         datetime
 * Purpose:        Functions to process date and time
 * Author:         John Thingstad
 * Created:        30/9/91
 *-------------------------------------------------------------------------
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include "datetime.h"
#include "validate.h"

/*
 * Function:       time_to_minutes
 * Purpose:        Given a time compute the number of minutes commensed since
 *                 midnight.
 * Arguments:      time structure
 * Returns:        number of minutes
 */
int time_to_minutes(const struct u_time *time)
{
  return time->hour * 60 + time->min;
}

/*
 * Function:       minutes_to_time
 * Purpose:        Given a number of minutes since midnight compute
 *                 the time.
 * Arguments:      minutes
 * Returns:        time structure
 */
struct u_time minutes_to_time(int min)
{
  struct u_time time;

  time.hour = min / 60;
  time.min  = min % 60;

  return time;
}

/*
 * Function:       normalize_year
 * Purpose:        Years are in two forms yyyy and yy. This function
 *                 converts all years to the form yyyy.
 * Arguments:      Year
 * Returns:        Year
 * NOTE:           only works reliably from year 1980 to year 2080
 */
int normalized_year(int year)
{
  assert(year >= 0 && year <= 9999);
 if (year < 99) /* year is a two digit number */
 {
  if (year > 80)
    year += 1900;
  else
    year += 2000;
 }

 return year;
}

/*
 * Function:      leap_year
 * Purpose:       Determine if given year is a leap year.
 * Arguments:     Year
 * Returns:       1 or 0
 */
int leap_year(int year)
{
 int leap;

 /* mostly to weed out bugs in other functions */
 assert(year >= 1980);

 /* We have leap year every 4'th year, except every 100'th year,
  * and every 400'th year. (Gregorian calender)
  */
 if ((year % 4 == 0) && ((year % 100 != 0) || (year % 400 != 0)))
   leap = 1;
 else
   leap = 0;

 return leap;
}

/*
 * Function:      date_to_day_nr
 * Purpose:       Given a date compute the days that have commensed since
 *                the start of that year.
 * Arguments:     date structure
 * Returns:       days commensed since the start of the year
 */
int date_to_day_nr(const struct u_date *date)
{
 /* number of days in a month */
 static const unsigned int jan = 31;
        const unsigned int feb = 28 + leap_year(date->year);
 static const unsigned int mar = 31;
 static const unsigned int apr = 30;
 static const unsigned int may = 31;
 static const unsigned int jun = 30;
 static const unsigned int jul = 31;
 static const unsigned int aug = 31;
 static const unsigned int sep = 30;
 static const unsigned int oct = 31;
 static const unsigned int nov = 30;

 /* For each month in the given year calculate the number of days into
  * that year that have passed when that month begins.
  */
 unsigned int total_days[12];

 total_days[ 0] = 0;
 total_days[ 1] = jan;
 total_days[ 2] = total_days[ 1] + feb;
 total_days[ 3] = total_days[ 2] + mar;
 total_days[ 4] = total_days[ 3] + apr;
 total_days[ 5] = total_days[ 4] + may;
 total_days[ 6] = total_days[ 5] + jun;
 total_days[ 7] = total_days[ 6] + jul;
 total_days[ 8] = total_days[ 7] + aug;
 total_days[ 9] = total_days[ 8] + sep;
 total_days[10] = total_days[ 9] + oct;
 total_days[11] = total_days[10] + nov;

 assert(date->month >= 1 && date->month <= 12);

 return total_days[date->month - 1] + date->day;
}

/*
 * Function:      day_nr_to_date
 * Purpose:       Given a year and a number of days passed since the start of
 *                that year compute the month and the day in that month
 *                then return day, month and year in date structure.
 * Arguments:     year and number of days
 * Returns:       date structure
 */
struct u_date day_nr_to_date(int year, int day_nr)
{
 struct u_date date = {0, 0, 0};
 int i;

 /* number of days in a month */
 static const unsigned int jan = 31;
        const unsigned int feb = 28 + leap_year(year);
 static const unsigned int mar = 31;
 static const unsigned int apr = 30;
 static const unsigned int may = 31;
 static const unsigned int jun = 30;
 static const unsigned int jul = 31;
 static const unsigned int aug = 31;
 static const unsigned int sep = 30;
 static const unsigned int oct = 31;
 static const unsigned int nov = 30;

 /* For each month in the given year calculate the number of days into
  * that year that have passed when that month begins.
  */
 unsigned int total_days[12];

 total_days[ 0] = 0;
 total_days[ 1] = jan;
 total_days[ 2] = total_days[ 1] + feb;
 total_days[ 3] = total_days[ 2] + mar;
 total_days[ 4] = total_days[ 3] + apr;
 total_days[ 5] = total_days[ 4] + may;
 total_days[ 6] = total_days[ 5] + jun;
 total_days[ 7] = total_days[ 6] + jul;
 total_days[ 8] = total_days[ 7] + aug;
 total_days[ 9] = total_days[ 8] + sep;
 total_days[10] = total_days[ 9] + oct;
 total_days[11] = total_days[10] + nov;

 assert(day_nr >= 1 && day_nr <= 366); /* 366 days in leap year */

 for (i = 0; i < 12; i++)
   if (day_nr - total_days[i] <= 31) break;

 date.day   = day_nr - total_days[i];
 date.month = i + 1;
 date.year  = year;

 return date;
}

/*
 * Function:      date_to_days_since_1980
 * Purpose:       As the name says given a date convert it to days elapsed
 *                since 1/1 1980.
 * Arguments:     date
 * Returns:       days
 */
unsigned long date_to_days_since_1980(struct u_date *date)
{
 unsigned long days = 0;
 int cur_year, years;
 int i;

 /* first convert the years passed since 1980 */
 years = date->year - 1980;

 /* primitive yes.. but intuetive */
 cur_year = 1980;
 for (i = 0; i < years; i++)
 {
  days += 365 + leap_year(cur_year);
  cur_year++;
 }

 /* now add the days into that year */
 days += date_to_day_nr(date);

 return days;
}

/*
 * Function:      date_time_to_minutes
 * Purpose:       given date and time as defined by structures u_time and u_date
 *                compute the numer of minutes elapsed since January 1'st
 *                1980 at time 00:00 GMT.
 *                Date and time are assumed to be GMT. (Greenwich Mean Time)
 * Arguments:     date and time
 * Returns:       minutes elapsed (i_time)
 */
i_time date_time_to_minutes(struct u_date *date, struct u_time *time)
{
 i_time minutes;
 unsigned long days;

 /* first convert the date to days since 1/1 1980 .. */
 days = date_to_days_since_1980(date);
```

```
  /* .. then convert to minutes .. */
  minutes = days * 24 * 60;

  /* .. and add in the minutes into that day (since midnight) */
  minutes += time_to_minutes(time);

  return minutes;
}

/*
 * Function:      minutes_to_days
 * Purpose:       convert minutes to days
 * Arguments:     number of minutes
 * Returns:       number of days, remaing minutes
 */
void minutes_to_days(i_time minutes, int *days, int *rest)
{
  *days = (int) (minutes / (60 * 24));
  *rest = (int) (minutes % (60 * 24));
}

/*
 * Function:      days_to_years
 * Purpose:       Given days since 1/1 1980 convert to years
 * Arguments:     number of days
 * Returns:       number of years, remaining days
 */
void days_to_years(int days, int *year, int *rest)
{
  *year = 1980;
  while (days > (365 + leap_year(*year)))
  {
    days -= (365 + leap_year(*year));
    (*year)++;
  }
  *rest = days;
}

/*
 * Function:      minutes_to_v_time
 * Purpose:       convert number of minutes since 1/1 1980 00:00 GMT
 *                to v_time structure that is year, day_nr, min_nr.
 * Arguments:     minutes
 * Returns:       v_time
 */
struct v_time minutes_to_v_time(i_time minutes)
{
  struct v_time time;
  int days;

  /* first strip into days and minutes into day */
  minutes_to_days(minutes, &days, &time.min_nr);

  /* We now have days since 1/1 1980.
   * Convert it to years and days
   */
  days_to_years(days, &time.year, &time.day_nr);

  return time;
}
```

```
/*
 * Function:      v_time_to_minutes
 * Purpose:       Conver a time on v_time form to minutes since
 *                1/1 1980 00:00 GMT.
 * Arguments:     pointer to v_time
 * Returns:       number of minutes
 */
i_time v_time_to_minutes(struct v_time *time)
{
  int years, days = 0;
  i_time minutes;
  int i;
  int cur_year;

  /* first convert the years passed since 1980 */
  years = time->year - 1980;

  /* primitive yes.. but intuetive */
  cur_year = 1980;
  for (i = 0; i < years; i++)
  {
    days += 365 + leap_year(cur_year);
    cur_year++;
  }

  /* add in days into year */
  days += time->day_nr;

  /* convert to minutes */
  minutes = ((long) days) * 24 * 60;

  /* and add in minutes into day */
  minutes += time->min_nr;

  return minutes;
}

/*
 * Function:      day_difference
 * Purpose:       Compute current time - start time in days
 * Arguments:     time1, time2 both in minutes since 1/1 1980 00:00 GMT
 * Returns:       days
 */
int day_difference(i_time time1, i_time time2)
{
  i_time diff = time2 - time1;
  int days;
  int rest;

  /* now convert to days */
  minutes_to_days(diff, &days, &rest);

  return days;
}

/***************************************************************************/
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
/*drafdrif.c*/
#include <stdlib.h>
```

```
#include <stdio.h>
#include "globals.h"
#include "draft.h"
#include "drift.h"

void main(void)
{
 double draft_mean;
 struct DRIFT_data dr;

 draft_mean = draft();
 dr = drift();

 if (dr.have_drift)
 {
   printf("\nThe total volume per day is about %8.2lf\n",
     dr.per_day * draft_mean
   );
   printf("\nThe total for this period is about %8.2f\n",
     dr.total * draft_mean
   );
 }
}
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
/*draft.h*/
/*
 * Header file for draft.c
 */

double draft(void);
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
/*draft.c*/
/*
 * Module:        draft.c
 * Purpose:       Rewrite of Kjell Olssons program
 *                to make it more modular and more stable.
 *                Basicly apply's KISS (Keep It Short and Simple) and
 *                AID (Abort if Ill Defined) methodology.
 *                It reduses the number of global variables.
 *                It apply's a 4:th order Runge-Kutta method to solve for draft.
 *                A new exponential function for temperature is used.
 * Author:        John Thingstad
 * Created:       19/9-1991
 */

#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <stdarg.h>
#include <math.h>
#include <time.h>
#include <signal.h>
#include <limits.h>
#include <values.h>
#include <string.h> /* called strings.h in UNIX environment */
#include "globals.h"
#include "datetime.h"
#include "uls.h"
#include "temp.h"
```

```
#include "airpr.h"
#include "prompt.h"
#include "draftout.h"

/* #define DEBUG here to activate log file */
/* #define DEBUG */


/*
 * Prototypes
 *
 */

void sig_break(void);

/* log file functions */
void Init_log_file(void);
int file_log(char *, ...);


/* Compute draft functions */

/***************************************************************************/

/*
 * Function:      sig_break
 * Purpose;       abort program gracefully
 * Arguments:     none
 * Returns:       none
 */
void sig_break(void)
{
#ifdef DEBUG
 file_log("\tCtrl-C pressed.\n\tProgram aborted.");
#endif
 exit(0);
};

#ifdef DEBUG
/*-------------------------------------------------------------------------
 * Functios associated with log file.
 * Used to log debugging information at various points.
 * Interface:
 *          void Init_log_file(void)
 *          int file_log(char *, ...)        -- uses standard printf format
 * NOTE: Init_log_file must be called prior to any call to log.
 *-------------------------------------------------------------------------
 */

FILE *log_file;

void Init_log_file(void)
{
 if ((log_file = fopen("polar.log", "w")) == NULL)
 {
  perror("Opening file polar.log");
 }
}

int file_log(char *format, ...)
{
 va_list arg_ptr;
 int count;

 /* make sure format string is not NULL or "" */
 assert(format && *format);
 assert(log_file != NULL);

 va_start(arg_ptr, format);

 count =  fprintf(log_file, "--> ");
 count = vfprintf(log_file, format, arg_ptr);
 count =  fprintf(log_file, "\n");
 if (count == EOF)
 {
   perror("Writing to file polar.log");
 }

 va_end(arg_ptr);

 return count;
}

/****************************************************************************/
#endif

/* Average monthly temperatures at the ice subsurface throughout a year.
 * Depends on salinity. Values taken from Sverdrup 1970 The Oceans.
 * Can be adjusted if better values become available.
 */
PRIVATE double SUR_TEMP_array[12] =
 {
 -1.75, /* jan */
 -1.75, /* feb */
 -1.75, /* mar */
 -1.75, /* apr */
 -1.35, /* may */
 -1.35, /* jun */
 -1.35, /* jul */
 -1.35, /* aug */
 -1.35, /* sep */
 -1.75, /* oct */
 -1.75, /* nov */
 -1.75  /* dec */
 };

struct SUR_TEMP_data
{
 i_time time;
 double temp;
};

struct SUR_TEMP_data get_SUR_TEMP_data(i_time time)
{
 struct SUR_TEMP_data data;
 struct v_time t_time;
 struct u_date date;

 /* convert time (minutes since 1/1 1980 00:00 GMT) to date */
 t_time = minutes_to_v_time(time); /* t_time = year, day_nr, min_nr */
 date = day_nr_to_date(t_time.year, t_time.day_nr);
 data.temp = SUR_TEMP_array[date.month - 1];

 date.day = 1;
 t_time.day_nr = date_to_day_nr(&date);
 t_time.min_nr = 0;

 data.time = v_time_to_minutes(&t_time);

 return data;
}

/*****************************************************************************/

/*
 * Function:        rad
 * Argument:        degrees
 * Purpose:         convert degrees to radians
 * Returns:         radians
 */
double rad(double degrees)
{
 static const double deg_to_rad = 57.29577951;

 return degrees / deg_to_rad;
}

/*
 * Function:        pas
 * Arguments:       mBar
 * Purpose:         convert millibar to pascal
 * Returns:         value in pascal
 */
double pas(double mBar)
{
 static const double mBar_to_pas = 100.0;

 return mBar * mBar_to_pas;
}

/*
 * Function:        transmit_time
 * Purpose:         Given the two sonar beacon recever counts compute
 *                  the actual transmission time in seconds.
 * Arguments:       count1, cont2 = the sonar beacon counts
 * Returns:         the computed transmit time
 */
double transmit_time(int count1, int count2)
{
 /*vvvvvvvvvvvvvvv   variables to be adjusted for each ULS bouy   vvvvvvvvvvvvvv*/
 /* gain for response time measurement */
 static const double CMI_c1 = 0.0000512;
 /* offset for resposetime measurement */
 static const double CMI_c2 = 0.00004;

 return 0.25 * ( (double) count1 + count2 ) * CMI_c1 + CMI_c2;
}

/*
 * Function:        water_pressure
 * Purpose:         Given water pressure transduser count compute the actual
 *                  water pressure in pascal.
 * Arguments:     tdcount = pressure transduser count
 * Returns:       the water pressure [pas]
 */
double water_pressure(int tdcount)
{
 /*vvvvvvvvvvvvvvv  variables to be adjusted for each ULS bouy  vvvvvvvvvvvvvv*/
 static const double press_c1 = 195.494;
 static const double press_c2 = -4369.6;
 /* digiquartz 8060 D. C. used for calibration */

 return ((double) tdcount) * press_c1 + press_c2;
}

/*
 * Function:      sensor_depth
 * Arguments:     water pressure, air pressure
 * Purpose:       Calculate water depth [m] given air pressure [pas]
 *                and water pressure [pas].
 * Retuns:        water debth [m]
 */
double sensor_depth(double water_pressure, double air_pressure)
{
 /*vvvvvvvvvvvvvvv  variables to be adjusted for each ULS location  vvvvvvvvvvv*/
 static const double g  = 9.829;  /* [m/s2]  gravity at 75N 12W */
 static const double ro = 1026.0;  /* [kg/m3] density of sea water */

 return (water_pressure - air_pressure) / (ro * g);
}

/*
 * Function:      sonar_depth
 * Arguments:     pressure sensor depth
 * Purpose:       given sensor depth compute sonar depth
 * Returns:       sonar depth
 */
double sonar_depth(double sensor_depth)
{
 /*vvvvvvvvvvvvvvv  variable to be adjusted for each ULS bouy  vvvvvvvvvvvvvv*/
 static const double dist_to_sonar = 0.13; /* [m] */
 return sensor_depth - dist_to_sonar;
}

/*
 * Function:      water_sound_speed
 * Arguments:     temperature [C], salinity [per mille], debth [m]
 * Purpose:       Compute the sound speed in water at a given debth
 *                with water a given temerature and with given salinity.
 * Returns:       sound speed [m/s]
 */
double water_sound_speed(float temp, double salinity, double depth)
{
 /*vvvvvvvvvvvvvvv  variable maybe to be adjusted  vvvvvvvvvvvvvv*/
/* the sound speed [m/s] in surface water at 0 C and 35 per mille salinity */
 const double surface_sound_speed = 1449.0;

 double sound_speed;

 sound_speed = surface_sound_speed
             +    4.6 * temp
             + 0.055 * temp * temp
             + 0.0003 * pow(temp, 3)
```

```
                + (1.39 - 0.012 * temp) * (salinity - 35)
                + 0.017 * depth;

  return sound_speed;
}

/****************************************************************************/

struct point
{
  double x;
  double y;
};

struct exp_const
{
  double a;
  double b;
  double c;
};

/*
 * Function:      compute_exp_const
 * Arguments:     three pointers to structures containing points
 * Purpose:       Given three points (x1, y1), (x2, y2), (x3, y3)
 *                draw a exponential curve through these points.
 *                ie. solve
 *                y1 = a * exp( b * x1 ) + c
 *                y2 = a * exp( b * x2 ) + c
 *                y3 = a * exp( b * x3 ) + c
 *                for a, b and c
 * Returns:       structure containing a, b and c
 */
struct exp_const compute_exp_const(struct point *p1, struct point *p2,
                                   struct point *p3)
{
  struct exp_const e;

  e.b = (p3->x - p1->x) / (p1->x * (p2->x - p3->x))
     * log((p3->y - p1->y) / (p2->y - p1->y));
  e.a = (p2->y - p1->y) / (exp(e.b * p2->x) - exp(e.b * p1->x));
  e.c = p1->y - e.a * exp(e.b * p1->x);

  return e;
}

/*
 * Function:      exp_func
 * Arguments:     exponential constants, values
 * Purpose:       given a,b,c and x compute a * exp(b * x) + .c
 * Returns:       value
 */
double exp_func(struct exp_const *c, double x)
{
  return c->a * exp(c->b * x) + c->c;
}

/****************************************************************************/

/*
 * COMMENTARY
 *
 * Calculating the distance to the ice subsurface.
 *
 *  For calculating the thickness of the ice we use an upward looking sonar buoy
 *  moored under the seaice, for example in the East Greenland Current.
 * To calculate the distance to the ice we need to know the sound speed.
 * (distance = sound_speed * transmission time)
 * The sound speed in water depends on the temperature of the water,
 * the salinity of the water and depth below sea surface (really the water pressure).
 * The problem lies in finding a good approximation for the temperature
 * and depth.
 *  The temperature can be approximated to follow an exponential curve.
 * This gives a function for temperature as a function of depth.
 * Given temperaures at three levels below sea surface we draw an
 * exponential function through these points. Used points are one substitute for the
 * ice subsurface, one at shallow depth and one at deeper depth, below the surface.
 *  A 4:th order Runge-Kutta method is used to solve the
 * initilal value problem for depth below the ice subsurface.
 * (based on sonar transmit time)
 * I took into acount that differences in velocity will diffract the sound,
 * and that the buoy has a tilt angle, by using only the vertical vector
 * when calculating depth.
 * (And hence I assume the bottom of the ice is flat in an approx. 3 m radius) JT 0991
 */

PRIVATE struct exp_const exp_c;

double temp(double depth)
{
  /* given exp_c compute the temperature as function of depth */
  return exp_func(&exp_c, depth);
}

double rel_speed(double debth)
{
 /*vvvvvvvvvvvvvvv   variable maybe to be adjusted   vvvvvvvvvvvvvv*/
  static const double salinity = 33;
  double sound_speed;

  sound_speed = water_sound_speed(
                        temp(debth),
                        salinity,
                        debth
  );

  return sound_speed;
}

double distance(double start_time, double stop_time,
                       double start_depth, double start_angle, int n)
{
  int k;
  double prev_velocity;
  double velocity;
  double depth;
  double delta_distance;
  double delta_depth;
  double step;
  double angle;
  double prev_angle;
  double F1, F2, F3, F4;
  /* Find sound path using 4:th order Runge-Kutta method
   * to integrate over varying velocity.
   * (Velocity vary's with depth so the sound does not follow
   *  a straight line. n = v1/v2)
   */

  depth  = start_depth;
  step = ( stop_time - start_time ) / ( (double) n );

  prev_angle = start_angle;

  /* calculate prev_velocity ( = velocity in first iteration) */
  F1 = rel_speed(depth);
  F2 = rel_speed(depth + (step / 2.0) * F1);
  F3 = rel_speed(depth + (step / 2.0) * F2);
  F4 = rel_speed(depth + step * F3);
  prev_velocity = (F1 + 2.0 * F2 + 2.0 * F3 + F4) / 6.0;

  for (k = 0; k < n; k++)
  {
    F1 = rel_speed(depth);
    F2 = rel_speed(depth + (step / 2.0) * F1);
    F3 = rel_speed(depth + (step / 2.0) * F2);
    F4 = rel_speed(depth + step * F3);
    velocity = (F1 + 2.0 * F2 + 2.0 * F3 + F4) / 6.0;

    delta_distance = step * velocity;
    angle = velocity / prev_velocity * prev_angle;

    delta_depth = delta_distance * cos(angle);
    depth -= delta_depth;

    prev_angle = angle;
    prev_velocity = velocity;
  }

  return depth;
}

double compute_ice_draft(
  struct point *surface,
  struct point *shallow,
  struct point *deep,
  double transmit_time,
  double depth,
  double tilt
  )
{
  /* Compute coefficients in exponential function through the three
   * temperature points.
   */
  exp_c = compute_exp_const(surface, shallow, deep);

  return distance(0.0, transmit_time, depth, tilt, 10);
}

/*****************************************************************************/

struct STAT_info
{
  double min_value;
  double max_value;
  double sum;
  int samples;
};

PRIVATE struct STAT_info current;

PRIVATE FILE *STAT_temp_file = NULL;

void init_STAT(void)
{
  STAT_temp_file = tmpfile();
  if (STAT_temp_file == NULL)
  {
    perror("Creating temporary file");
    exit(1);
  }

  current.min_value = MAXDOUBLE;
  current.max_value = MINDOUBLE;
  current.sum = 0.0;
  current.samples = 0;
}

void update_STAT(struct DRAFT_data *data, double min_limit)
{
  assert(STAT_temp_file != NULL);

  if (data->draft > min_limit)
  {
    /* update max and min values */
    if (data->draft < current.min_value)
      current.min_value = data->draft;
    else
    if (data->draft > current.max_value)
      current.max_value = data->draft;

    current.sum += data->draft;
    current.samples++;

    /* store the draft temporarly on a file */
    fwrite(&(data->draft), sizeof(double), 1, STAT_temp_file);
    if (ferror(STAT_temp_file))
    {
      perror("Writing to temporary file");
      exit(1);
    }
  }
}

struct STAT_data get_STAT(void)
{
  struct STAT_data data;
  double samples = (double) current.samples;
  double draft;
  double average;
  double deviation_sum = 0.0;
  double variance_sum = 0.0;
  double skew_sum = 0.0;
  double curtosis_sum = 0.0;
  double P1, P2, P3, P4;
  int i;

  assert(STAT_temp_file != NULL);
  rewind(STAT_temp_file);

  data.samples   = current.samples;
  data.min_value = current.min_value;
  data.max_value = current.max_value;

  average = current.sum / samples;

  for (i = 0; i < current.samples; i++)
  {
    fread(&draft, sizeof(double), 1, STAT_temp_file);
    if (ferror(STAT_temp_file))
    {
      perror("Reading temporary file");
      exit(1);
    }

    P1 = fabs(draft - average); /* deviation */
    P2 = P1* P1;            /* deviation to the second power */
    P3 = P2 * P1;           /* deviation to the third power  */
    P4 = P3 * P1;           /* deviation to the fourth power */

    deviation_sum     += P1;
    variance_sum      += P2;
    skew_sum          += P3;
    curtosis_sum      += P4;
  }

  data.average = average;
  data.mean_deviation = deviation_sum / samples;

  if (samples > 0)
    data.varians = variance_sum / (samples - 1.0);
  else
    data.varians = 0.0;
  data.no_varians = (data.varians == 0.0 ? true : false);

  data.standard_deviation = sqrt(data.varians);

  if (data.no_varians == false)
  {
    data.skew =
      skew_sum /
      (samples * data.varians * data.standard_deviation);
    data.curtosis =
      curtosis_sum / (samples * pow(data.standard_deviation, 2) - 3.0);
  }

  fclose(STAT_temp_file);
  return data;
}

/****************************************************************************/

double draft(void)
{
  i_time start_time;
  i_time  stop_time;
  i_time  cur_time;
  i_time   delta_t = 4;

  struct u_date usr_start_date = {  1,  1, 88 };
  struct u_time usr_start_time = {  0,  0 };
  struct u_date usr_stop_date  = { 1,  1, 88 };
  struct u_time usr_stop_time  = { 1, 0 };

  struct   ULS_data ULS;
  struct TEMP_data_header S_TEMP_header;
  struct TEMP_data_header D_TEMP_header;
  struct S_TEMP_data S_TEMP;
  struct D_TEMP_data D_TEMP;
  struct AIR_PR_data AIR_PR;

  struct SUR_TEMP_data SUR_TEMP;

  struct point surface;
  struct point shallow;
  struct point deep;

  double trans_time;
  double depth;
  double draft;
  int out_of_range;
  double minimum_value;

  struct DRAFT_data data;
  struct STAT_data  stat;
  struct EXTRA_data extra;

  if (signal(SIGINT, sig_break) == SIG_ERR)
  {
    perror("Signal failed");
    exit(0);
  }

#ifdef DEBUG
  Init_log_file();
#endif

  puts("");
  puts("(c) Norwegian Polar Research Institute");
  puts("Program to calculate ice draft and ice drift.");
  puts("Ctrl-C aborts program.");
  puts("");

  puts("Enter draft start time:");
  usr_start_date = prompt_date(&usr_start_date);
  usr_start_time = prompt_time(&usr_start_time);
  puts("");

  start_time = date_time_to_minutes(&usr_start_date, &usr_start_time);

  puts("Enter draft stop time:");
  usr_stop_date = prompt_date(&usr_stop_date);
  usr_stop_time = prompt_time(&usr_stop_time);
  puts("");

  stop_time = date_time_to_minutes(&usr_stop_date, &usr_stop_time);
```

```
puts("Enter the minimum ice thickness to be accepted");
minimum_value = prompt_real(2.0);
puts("");

/*vvvvvvvvvvvvvv  paths maybe to be adjusted  vvvvvvvvvvvvv*/
puts("Enter ULS file directory:");
  init_ULS_file(prompt_file("C:\\TC\\TCPROG\\DATA\\"),
                start_time);
puts("");
puts("Enter full shallow temperature file name:");
init_S_TEMP_file(prompt_file("C:\\TC\\TCPROG\\DATA\\V1281.DAT"),
                start_time);
puts("");
puts("Enter full deep temperature file name:");
init_D_TEMP_file(prompt_file("C:\\TC\\TCPROG\\DATA\\T981.DAT"),
                start_time);
puts("");
puts("Enter full air pressure file name:");
init_AIR_PR_file(prompt_file("C:\\TC\\TCPROG\\DATA\\LTR8788.DAT"),
                start_time);

S_TEMP_header = get_S_TEMP_header();
D_TEMP_header = get_D_TEMP_header();

if (
S_TEMP_header.lon  != D_TEMP_header.lon ||
S_TEMP_header.lat  != D_TEMP_header.lat
)
{
  fputs("\nTemperature files must be from the same location.\n", stderr);
  exit(1);
}

init_DRAFT_file();

init_STAT();

puts("");
init_STAT_file();

cur_time = start_time;
out_of_range = 0;
while (cur_time <= stop_time)
{
  ULS    =   get_ULS_data(cur_time);
  S_TEMP = get_S_TEMP_data(cur_time);
  D_TEMP = get_D_TEMP_data(cur_time);
  AIR_PR = get_AIR_PR_data(cur_time);

  SUR_TEMP = get_SUR_TEMP_data(cur_time);

  if (ULS.cnt_1 <= 1952 && ULS.cnt_2 <= 1952 &&
          fabs(ULS.cnt_2 - ULS.cnt_1) < 40)
  {
    trans_time = transmit_time(ULS.cnt_1, ULS.cnt_2);

    depth =
          sonar_depth(
            sensor_depth(
              water_pressure(ULS.press_transd_cnt),
            pas(AIR_PR.pressure)
           )
          );

    /*vvvvvvvvvvvvvvv  variables to be adjusted for each ULS location   vvvvvvvvvv*/
     surface.x = 1;  surface.y = SUR_TEMP.temp;
     shallow.x = 72;  shallow.y = S_TEMP.temp;
          deep.x = 101;   deep.y = D_TEMP.temp;

     draft =
          compute_ice_draft(&surface, &shallow, &deep,
                            trans_time,
                            depth,
                            rad(ULS.angle)
          );

     data.time      = cur_time;
     data.draft     = draft;
     data.approx    = ULS.approx;
     data.debth     = depth;

     update_STAT(&data, minimum_value);
     put_DRAFT_data(&data);
    }
    else
    {
     out_of_range++;
    }

    cur_time += delta_t;
  }

  stat = get_STAT();

  extra.start_time  = start_time;
  extra.stop_time   = stop_time;
  extra.longtitude  = S_TEMP_header.lon;
  extra.latitude    = S_TEMP_header.lat;
  extra.lower_limit = minimum_value;
  extra.fall_out    = out_of_range;

  put_STAT_data(&stat, &extra);

  return stat.average;
}
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
/*temp.h*/
/*
 * Header file for TEMP.C
 */

#ifndef TEMP
#define TEMP

struct TEMP_data_header
{
  i_time time;
  double lat;                 /* latitude buoy is placed */
  double min_lat;  /* uncertainty ? */
  double lon;                 /* lontitude buoy is placed */
  double min_lon; /* uncertainty ? */
  int depth;        /* sensor depth */
};

struct S_TEMP_data
{
  i_time time;
  double uc;
  double vc;
  double direction;
  double speed;
  double temp;
};

struct D_TEMP_data
{
  i_time time;
  double temp;
};

/* Header data interface */
struct TEMP_data_header get_S_TEMP_header(void);
struct TEMP_data_header get_D_TEMP_header(void);

/* S_TEMP interface */
void init_S_TEMP_file(const char *name, i_time time);
struct S_TEMP_data get_S_TEMP_data(i_time time);

/* D_TEMP interface */
void init_D_TEMP_file(const char *name, i_time time);
struct D_TEMP_data get_D_TEMP_data(i_time time);

#endif
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
/*temp.c*/
/*-------------------------------------------------------------------------
 * Module:       TEMP.C
 * Purpose:   Functions associated with reading S_TEMP and D_TEMP files
 * Created:      1/10/91
 *-------------------------------------------------------------------------
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include "globals.h"
#include "datetime.h"
#include "temp.h"

/* When doing maintenance #define DEBUG to activate modules
 * auto integrity check.
 */
/* #define DEBUG */

/*
 * COMMENTARY
 *
 * temp file formats
 *
 * The temp files are structured as follows:
 *         header:
 *         line:
 *
 *  header:
 *         1) station
 *         2) latitute and longtitude
 *         3) date etc.
 *         4) bathymetric waterdepth
 *         5) max debth of buoy
 *         6) mnemonic of line format
 *         7) resyme of equipment and time intervall etc.
 *         8) # (header termination symbol)
 *
 *  line:
 *         S_TEMP_file:
 *         D_TEMP_file:
 *
 *  S_TEMP_file:
 *         DAY UC VC DIR SPD TEMP
 *
 *         DAY     = days since measurement started  (int)
 *         UC      = north south current component    (float)
 *         VC      = east  west  current component    (float)
 *         DIR   = current direction          (float)
 *         SPD     = current speed                     (float)
 *         TEMP    = water temperature                 (float)
 *
 *  D_TEMP_file:
 *         DAY TEMP
 *
 *         DAY     = days since mesurement started   (int)
 *         TEMP    = water temperature                 (float)
 */

PRIVATE struct TEMP_pack_header
{
 double lat;                  /* latitudeof  buoy position */
 double min_lat;  /* minutes of latitude - uncertainty ? */
 double lon;                  /* lontitude of  buoy position*/
 double min_lon;  /* minutes of longitude - uncertainty ? */
 int depth;             /* sensor depth */
 int day;           /* date of first day of mesurement */
 int month;
 int year;
};

PRIVATE struct S_TEMP_pack
{
 int day;          /* number of days since mesurements started */
 double uc;                   /* north-south coordinate  */
 double vc;                   /* east-west   coordinate  */
 double direction;/* current direction (deg) */
 double speed;                /* current speed      */
 double temp;        /* water tempeture a shallow depth */
};

PRIVATE struct D_TEMP_pack
{
 int day;          /* number of days since measurements started */
 double temp;                 /* water temperature at deeper depth*/
```

```
};

/*
 * Prototypes for functions local to this module
 */

PRIVATE bool read_TEMP_pack_header(FILE *file_ptr, const char *name,
                                   struct TEMP_pack_header *header);

/* low level S_TEMP functions */
PRIVATE bool find_S_TEMP_pack(const int day);
PRIVATE bool read_S_TEMP_pack(struct S_TEMP_pack *pack);

/* low level D_TEMP functions */
PRIVATE bool find_D_TEMP_pack(const int day);
PRIVATE bool read_D_TEMP_pack(struct D_TEMP_pack *pack);

/* header functions */
PRIVATE void TEMP_pack_header_to_data(struct TEMP_pack_header *pack,
            i_time time, struct TEMP_data_header *data);
PRIVATE bool read_TEMP_data_header(FILE *file_ptr, const char *name,
                                   struct TEMP_data_header *data);
PRIVATE void init_TEMP_header(FILE *file_ptr, const char *name,
                             struct TEMP_data_header *header);

/* time conversion routines */
PRIVATE int time_to_file_day(i_time request_time,
             struct TEMP_data_header *header);
PRIVATE i_time file_day_to_time(int day, struct TEMP_data_header *header);

/* high level S_TEMP routines */
PRIVATE bool find_S_TEMP_data(i_time request_time);
PRIVATE void S_TEMP_pack_to_data(struct S_TEMP_pack *pack, i_time time,
                                 struct S_TEMP_data *data);
PRIVATE bool read_S_TEMP_data(struct S_TEMP_data *data);
PRIVATE struct S_TEMP_data copy_S_TEMP_data(const struct S_TEMP_data *data2);

/* high level S_TEMP routines */
PRIVATE bool find_D_TEMP_data(i_time request_time);
PRIVATE void D_TEMP_pack_to_data(struct D_TEMP_pack *pack, i_time time,
                                 struct D_TEMP_data *data);
PRIVATE bool read_D_TEMP_data(struct D_TEMP_data *data);
PRIVATE struct D_TEMP_data copy_D_TEMP_data(const struct D_TEMP_data *data2);

/*
 * Data local to this module
 */
PRIVATE char   S_TEMP_name[80];
PRIVATE FILE  *S_TEMP_file = NULL;
PRIVATE struct TEMP_data_header S_TEMP_header;
PRIVATE struct S_TEMP_data S_TEMP_array[2];

PRIVATE char   D_TEMP_name[80];
PRIVATE FILE  *D_TEMP_file = NULL;
PRIVATE struct  TEMP_data_header D_TEMP_header;
PRIVATE struct D_TEMP_data D_TEMP_array[2];

/*------------------------------------------------------------------------
 * Function to read file header information in both
 * deep temperature file (D_TEMP) and shallow temerature file (S_TEMP)
 *------------------------------------------------------------------------
 */

/*
 * Function:       read_TEMP_pack_header
 * Arguments:      file pointer, full file name, pointer to header structure
 * Purpose:        Read the header info from the given file into the given
 *                 structure.
 * Returns:        true if successful else false
 */
bool read_TEMP_pack_header(FILE *file_ptr, const char *name,
                                   struct TEMP_pack_header *header)
{
  char buffer[82];
  char *result;
  char chr;

  rewind(file_ptr);

  if (
    fgets(buffer, 80, file_ptr) == NULL
  )
  {
    if (feof(file_ptr))
      return false;
    else
    {
      perror(name);
      exit(1);
    }
  }

  result = fgets(buffer, 80, file_ptr);
  if (result == NULL)
  {
    if (feof(file_ptr))
      return false;
    else
    {
      perror(name);
      exit(1);
    }
  }

  header->lat      = strtod(buffer, &result);
  header->min_lat  = strtod(result, &result);
  header->lon      = strtod(result, &result);
  header->min_lon  = strtod(result, &result);


  result = fgets(buffer, 80, file_ptr);
  if (result == NULL)
  {
    if (feof(file_ptr))
      return false;
    else
    {
      perror(name);
      exit(1);
    }
  }
```

```
if (
  sscanf(buffer, "%*d %d %d %d",
    &header->month,
    &header->year,
    &header->day
  ) == EOF
)
{
  fprintf(stderr, "\n%s: %s\n", name, "Invalid format");
  exit(1);
}

result = fgets(buffer, 80, file_ptr);
if (result == NULL)
{
  if (feof(file_ptr))
    return false;
  else
  {
    perror(name);
    exit(1);
  }
}

result = fgets(buffer, 80, file_ptr);
if (result == NULL)
{
  if (feof(file_ptr))
    return false;
  else
  {
    perror(name);
    exit(1);
  }
}

if (sscanf(buffer, "%d", &header->depth) == EOF)
{
  fprintf(stderr, "\n%s: %s\n", name, "Invalid format");
  exit(1);
}

do
{
  result = fgets(buffer, 80, file_ptr);
  if (result == NULL)
  {
    if (feof(file_ptr))
        return false;
    else
    {
        perror(name);
        exit(1);
    }
  }

  if (
    sscanf(buffer, "%c", &chr) == EOF
  )
  {
    fprintf(stderr, "\n%s: %s\n", name, "Invalid format");
```

```
      exit(1);
    }
  } while (chr != '#');

  return true;
}


/*------------------------------------------------------------------------
 * Functions associated with reading shallow temperature file
 *------------------------------------------------------------------------
 */

/*
 * Function:       find_S_TEMP_pack
 * Arguments:      day
 * Purpose:        given a day (since start date) find S_TEMP info for that day
 * Returns:        true if successful else false
 */
bool find_S_TEMP_pack(const int day)
{
  char buffer[82];
  char *result;
  int search_day;
  bool found = false;
  long pos;

  assert(S_TEMP_file != NULL);

  while (!feof(S_TEMP_file) && found == false)
  {
    result = fgets(buffer, 80, S_TEMP_file);
    if (result == NULL)
    {
      if (feof(S_TEMP_file))
            return false;
      else
      {
            perror(S_TEMP_name);
            exit(1);
      }
    }

    if (
      sscanf(buffer, "%d", &search_day) == EOF
    )
    {
      fprintf(stderr, "\n%s: %s\n", S_TEMP_name, "Invalid format\n");
      exit(1);
    }

    if (search_day >= day)
      found = true;
  }

  /* We read past the packet we found ...
   * Reset file to position we had before we read the last line in order
   * to let read_S_TEMP_pack read the packet.
   */
  pos = -((long) strlen(buffer) + 1);
  fseek(S_TEMP_file, pos, SEEK_CUR);
  return found;
}

/*
 * Function:       read_S_TEMP_pack
 * Arguments:      pointer to S_TEMP_pack
 * Purpose:        read the next S_TEMP_pack for the file
 * Returns:        true if successful else false
 */
bool read_S_TEMP_pack(struct S_TEMP_pack *pack)
{
  char buffer[82];
  char *result;

  assert(S_TEMP_file != NULL);

  result = fgets(buffer, 80, S_TEMP_file);
  if (result == NULL)
  {
    if (feof(S_TEMP_file))
      return false;
    else
    {
      perror(S_TEMP_name);
      exit(1);
    }
  }

  if (
    sscanf(buffer, "%d %lf %lf %lf %lf %lf",
      &pack->day,
      &pack->uc,
      &pack->vc,
      &pack->direction,
      &pack->speed,
      &pack->temp)
    == EOF
  )
  {
    fprintf(stderr, "\n%s: %s\n", S_TEMP_name, "Invalid format");
    exit(1);
  }

  return true;
}

/*-----------------------------------------------------------------------
 * Functions associated with reading deep temperature file
 *-----------------------------------------------------------------------
 */

/*
 * Function:       find_D_TEMP_pack
 * Arguments:      day
 * Purpose:        given a day (since start date) find the closest file entry
 * Returns:        true if successful else false
 */
bool find_D_TEMP_pack(const int day)
{
  char buffer[82];
  char *result;
  int search_day;
  bool found = false;
  long pos;

  assert(D_TEMP_file != NULL);

  while (!feof(S_TEMP_file) && found == false)
  {
    result = fgets(buffer, 80, D_TEMP_file);
    if (result == NULL)
    {
      if (feof(D_TEMP_file))
              return false;
      else
      {
              perror(D_TEMP_name);
              exit(1);
      }
    }

    if (
      sscanf(buffer, "%d", &search_day) == EOF
    )
    {
      fprintf(stderr, "\n%s: %s\n", D_TEMP_name, "Invalid format\n");
      exit(1);
    }

    if (search_day >= day)
      found = true;
  }

  /* We read past the packet we found ...
   * Reset file to position we had before we read the last line in order
   * to let read_D_TEMP_pack read the packet.
   */
  pos = -((long) strlen(buffer) + 1);
  fseek(D_TEMP_file, pos, SEEK_CUR);

  return found;
}

/*
 * Function:       read_D_TEMP_pack
 * Arguments:      pointer to D_TEMP_pack
 * Purpose:        read the next D_TEMP_pack from the file
 * Returns:        true if successful else false
 */
bool read_D_TEMP_pack(struct D_TEMP_pack *pack)
{
  char buffer[82];
  char *result;

  result = fgets(buffer, 80, D_TEMP_file);
  if (result == NULL)
  {
    if (feof(D_TEMP_file))
      return false;
    else
    {
      perror(D_TEMP_name);
      exit(1);
    }
  }

  if (
    sscanf(buffer, "%d %lf", &pack->day, &pack->temp)
    == EOF
  )
  {
    fprintf(stderr, "\n%s: %s\n", D_TEMP_name, "Invalid format");
    exit(1);
  }

  return true;
}

/*--------------------------------------------------------------------------
 * Temperature file high level functions
 *--------------------------------------------------------------------------
 */

/*
 * Function:      TEMP_pack_header_to_data
 * Arguments:     pointer to TEMP_pack_header, system time,
 *                pointer to TEMP_data_header
 * Purpose:       create a TEMP_data_header from a TEMP_pack_header
 * returns:       none
 */
void TEMP_pack_header_to_data(struct TEMP_pack_header *pack, i_time time,
                              struct TEMP_data_header *data)
{
  data->time    = time;
  data->lat     = pack->lat;
  data->min_lat = pack->min_lat;
  data->lon     = pack->lon;
  data->min_lon = pack->min_lon;
  data->depth   = pack->depth;
}

/*
 * Function:      read_TEMP_data_header
 * Purpose:       read a TEMP_pack_header and convert time format to
 *                minutes since 1/1/1980 00:00 GMT.
 *                This gives a TEMP_data_header.
 * Arguments:     file pointer, name of file, pointer to TEMP_data_header
 * Returns:       true if success else false
 */
bool read_TEMP_data_header(FILE *file_ptr, const char *name,
                           struct TEMP_data_header *data)
{
  struct TEMP_pack_header pack;
  struct u_date date;
  struct u_time time;
  i_time system_time;
  bool result;

  result = read_TEMP_pack_header(file_ptr, name, &pack);
  if (result == true)
  {
    /* set date in date structure */
    date.year   = normalized_year(pack.year);
    date.month  = pack.month;
    date.day    = pack.day;

    /* assume midnight */
    time.hour = 0;
    time.min  = 0;

    /* get minutes since 1/1 1980 00:00 GMT */
    system_time = date_time_to_minutes(&date, &time);

    TEMP_pack_header_to_data(&pack, system_time, data);
  }

  return result;
}

/*
 * Function:       time_to_file_day
 * Arguments:      time    - time in minutes since 1/1 1980 00:00 GMT
 *                 pointer to TEMP_data_header
 * Purpose:        Temperature file has a header containing the starting date.
 *                 Subsequent entrys in the file contain the day since this
 *                 date starting with 1.
 *                 Convert request_time to days since start date in header.
 * Returns:        day index in file (it it exists in file else number < 1)
 */
int time_to_file_day(i_time request_time, struct TEMP_data_header *header)
{
  int day;

  /* get days since header date */
  day = day_difference(header->time, request_time);

  return day + 1;
}

/*
 * Function:       file_day_to_time
 * Purpose:        Given a day and a pointer to the file header compute
 *                 the time given in minutes since 1/1 1980 00:00 GMT.
 * Arguments:      day, pointer to header TEMP_data_header
 * Returns:        time in minutes
 */
i_time file_day_to_time(int day, struct TEMP_data_header *header)
{
  i_time time;

  /* first make days into minutes */
  time = ((long) day - 1) * 60 * 24;

  /* now take the header time and add this in */
  time += header->time;

  return time;
}

/*
 * Function:       init_TEMP_header
 * Purpose:        Read a TEMP_data_header from a file given the file pointer.
 * Arguments:      pointer to TEMP_data_header
 * Returns:          none
 */
void init_TEMP_header(FILE *file_ptr, const char *name,
                      struct TEMP_data_header *header)
{
  assert(file_ptr != NULL);

  if (read_TEMP_data_header(file_ptr, name, header) == false)
  {
    fprintf(stderr, "\n%s: %s\n", name, "Invalid header format");
    exit(1);
  }
}

/*------------------------------------------------------------------------
 * Shallow temperature file high level functions
 *------------------------------------------------------------------------
 */

/*
 * Function:         find_S_TEMP_data
 * Arguments:        system time (days since 1/1 1980 00:00 GMT)
 * Purpose:          given a system time request find the closest file entry
 * Returns:          true if successful else false
 */
bool find_S_TEMP_data(i_time request_time)
{
  int day;

  day = time_to_file_day(request_time, &S_TEMP_header);
  if (day < 1)
  {
    fprintf(stderr, "\n%s: %s\n", S_TEMP_name, "No data for that interval");
    exit(1);
  }

  return find_S_TEMP_pack(day);
}

/*
 * Function:         S_TEMP_pack_to_data
 * Arguments:        pointer to pack, system time, pointer to data
 * Purpose:          convert pack structure to data structure
 * Returns:          none
 */
void S_TEMP_pack_to_data(struct S_TEMP_pack *pack, i_time time,
                         struct S_TEMP_data *data)
{
  data->time      = time;
  data->uc        = pack->uc;
  data->vc        = pack->vc;
  data->direction = pack->direction;
  data->speed     = pack->speed;
  data->temp      = pack->temp;
}

/*
 * Function:         read_S_TEMP_data
 * Arguments:        pointer to S_TEMP_data structure
 * Purpose:          find the next S_TEMP file entry from file
 * Returns:          true if successful else false
 */
bool read_S_TEMP_data(struct S_TEMP_data *data)
{
  struct S_TEMP_pack pack;
  i_time time;
  bool result;

  result = read_S_TEMP_pack(&pack);
  if (result == true)
    time = file_day_to_time(pack.day, &S_TEMP_header);

  S_TEMP_pack_to_data(&pack, time, data);

  return result;
}

/*
 * Function:        init_S_TEMP_file
 * Argumets:        file name, start time (system format)
 * Purpose:         setup S_TEMP file manager
 * Returns:         none
 */
void init_S_TEMP_file(const char *name, i_time time)
{
  strcpy(S_TEMP_name, name);

  S_TEMP_file = fopen(S_TEMP_name, "r");
  if (S_TEMP_file == NULL)
  {
    perror(S_TEMP_name);
    exit(1);
  }

  /* read header info */
  init_TEMP_header(S_TEMP_file, S_TEMP_name, &S_TEMP_header);

  /* find first entry for corresponding time */
  if (find_S_TEMP_data(time) == false)
  {
    fprintf(stderr, "\n%s: %s\n", S_TEMP_name,
              " Unable to find data for that interval\n", stderr);
    exit(1);
  }

  /* initilize array */
  if (
    read_S_TEMP_data(&S_TEMP_array[0]) == false ||
    read_S_TEMP_data(&S_TEMP_array[1]) == false
  )
  {
    fprintf(stderr, "\n%s: %s\n", S_TEMP_name, "Unexpected EOF");
    exit(1);
  }
}

/*
 * Function:        copy_S_TEMP_data
 * Arguments:       pointer to S_TEMP_data structure
 * Purpose:         copy a S_TEMP data structure
 * Returns:         the copyed S_TEMP_data
 */
```

```
struct S_TEMP_data copy_S_TEMP_data(const struct S_TEMP_data *data2)
{
  struct S_TEMP_data data1;

  data1.time      = data2->time;
  data1.uc        = data2->uc;
  data1.vc        = data2->vc;
  data1.direction = data2->direction;
  data1.speed     = data2->speed;
  data1.temp      = data2->temp;

  return data1;
}

/*
 * Function:      get_S_TEMP_data
 * Argumets:      request time (system format)
 * Purpose:       find the S_TEMP_data structure with the time closest
 *                to the request time.
 * Returns:       the S_TEMP_data structure
 */
struct S_TEMP_data get_S_TEMP_data(i_time time)
{
  assert(S_TEMP_file != NULL);

  if (time >= S_TEMP_array[1].time)
  {
    do
    {
      /* The second time in the array is better that the first.
       * Make the second item the first and read a new second item.
       */
      S_TEMP_array[0] = copy_S_TEMP_data(&S_TEMP_array[1]);

      if (read_S_TEMP_data(&S_TEMP_array[1]) == false)
      {
            fprintf(stderr, "\n%s: %s\n", "Unexpected EOF");
            exit(1);
      }
    } while (time >= S_TEMP_array[1].time);
    /* repeat until this is no longer the case */
  }

  return S_TEMP_array[0];
}

/*
 * Function:      get_S_TEMP_header
 * Arguments:     none
 * Purpose:       get the S_TEMP header info.
 * Returns:       pointer to S_TEMP_header
 * NOTE:          init_S_TEMP_file must have been called prior to call to
 *                this function.
 */
struct TEMP_data_header get_S_TEMP_header(void)
{
  assert(S_TEMP_file != NULL);

  return S_TEMP_header;
}
```

```
/*-------------------------------------------------------------------------
 * Deep temperature file high level functions
 *-------------------------------------------------------------------------
 */

bool find_D_TEMP_data(i_time request_time)
{
  int day;

  day = time_to_file_day(request_time, &D_TEMP_header);
  if (day < 1)
  {
    fprintf(stderr, "\n%s: %s\n", D_TEMP_name, "No data for that interval");
    exit(1);
  }

  return find_D_TEMP_pack(day);
}

void D_TEMP_pack_to_data(struct D_TEMP_pack *pack, i_time time,
                                  struct D_TEMP_data *data)
{
  data->time      = time;
  data->temp      = pack->temp;
}

bool read_D_TEMP_data(struct D_TEMP_data *data)
{
  struct D_TEMP_pack pack;
  i_time time;
  bool result;

  result = read_D_TEMP_pack(&pack);
  if (result == true)
    time = file_day_to_time(pack.day, &D_TEMP_header);

  D_TEMP_pack_to_data(&pack, time, data);

  return result;
}

void init_D_TEMP_file(const char *name, i_time time)
{
  strcpy(D_TEMP_name, name);

  D_TEMP_file = fopen(D_TEMP_name, "r");
  if (D_TEMP_file == NULL)
  {
    perror(D_TEMP_name);
    exit(1);
  }

  /* read header info */
  init_TEMP_header(D_TEMP_file, D_TEMP_name, &D_TEMP_header);

  /* find first entry for corresponding time */
  if (find_D_TEMP_data(time) == false)
  {
    fprintf(stderr, "\n%s: %s\n", D_TEMP_name,
              " Unable to find data for that interval\n", stderr);
    exit(1);
  }

  /* initilize array */
  if (
    read_D_TEMP_data(&D_TEMP_array[0]) == false ||
    read_D_TEMP_data(&D_TEMP_array[1]) == false
  )
  {
    fprintf(stderr, "\n%s: %s\n", D_TEMP_name, "Unexpected EOF");
    exit(1);
  }
}

struct D_TEMP_data copy_D_TEMP_data(const struct D_TEMP_data *data2)
{
  struct D_TEMP_data data1;

  data1.time      = data2->time;
  data1.temp      = data2->temp;

  return data1;
}

struct D_TEMP_data get_D_TEMP_data(i_time time)
{
  assert(D_TEMP_file != NULL);

  if (time >= D_TEMP_array[1].time)
  {
    do
    {
      /* The second time in the array is better that the first.
       * Make the second item the first and read a new second item.
       */
      D_TEMP_array[0] = copy_D_TEMP_data(&D_TEMP_array[1]);

      if (read_D_TEMP_data(&D_TEMP_array[1]) == false)
      {
            fprintf(stderr, "\n%s: %s\n", "Unexpected EOF");
            exit(1);
      }
    } while (time >= D_TEMP_array[1].time);
    /* repeat until this is no longer the case */
  }

  return D_TEMP_array[0];
}

struct TEMP_data_header get_D_TEMP_header(void)
{
  assert(D_TEMP_file != NULL);

  return D_TEMP_header;
}

/****************************************************************************/

#ifdef DEBUG
void main(void)
{
  struct u_date start_date;
struct u_date stop_date;
struct u_time start_time;
struct u_time stop_time;
struct v_time cur_time;
i_time internal_time;
i_time delta_i;
i_time termination;
struct S_TEMP_data Sdata;
struct D_TEMP_data Ddata;

start_date.year = 88; start_date.month = 1; start_date.day = 1;
start_time.hour = 0;  start_time.min   = 0;

stop_date.year = 88; stop_date.month = 1; stop_date.day = 2;
stop_time.hour = 0;  stop_time.min   = 0;

delta_i = 4;

puts("Enter date to start sampling S_TEMP data");
start_date = prompt_date(&start_date);

puts("Enter time to start sampling S_TEMP data");
start_time = prompt_time(&start_time);

puts("Enter date to stop sampling S_TEMP data");
stop_date = prompt_date(&stop_date);

puts("Enter time to stop sampling S_TEMP data");
stop_time = prompt_time(&stop_time);

puts("Enter sampling frequency");
scanf("%d", &delta_i);

internal_time = date_time_to_minutes(&start_date, &start_time);
termination   = date_time_to_minutes(&stop_date, &stop_time);

init_S_TEMP_file("C:\\TC\\TCPROG\\DATA\\V1281.DAT", internal_time);
init_D_TEMP_file("C:\\TC\\TCPROG\\DATA\\T981.DAT",  internal_time);

while (internal_time < termination)
{
  Sdata = get_S_TEMP_data(internal_time);
  Ddata = get_D_TEMP_data(internal_time);

  cur_time = minutes_to_v_time(internal_time);
  printf("\nRequest time: %2.2d:%3.3d:%4.4d",
    cur_time.year,
    cur_time.day_nr,
    cur_time.min_nr
  );

  puts("\nShallow temperature file");
  cur_time = minutes_to_v_time(Sdata.time);
  printf("Time: %2.2d:%3.3d:%4.4d ",
    cur_time.year,
    cur_time.day_nr,
    cur_time.min_nr
  );
  printf("Data: %.2lf %.2lf %.2lf %.2lf %.2lf\n",
    Sdata.uc,
    Sdata.vc,
      Sdata.direction,
      Sdata.speed,
      Sdata.temp
    );

    puts("Deep temperature file");
    cur_time = minutes_to_v_time(Ddata.time);
    printf("Time: %2.2d:%3.3d:%4.4d ",
      cur_time.year,
      cur_time.day_nr,
      cur_time.min_nr
    );
    printf("Data: %.2lf\n",
      Ddata.temp
    );

    internal_time += delta_i;
  }
}
#endif
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
/*airpr.h*/
/*
 * header file for AIRPR.C
 */

#ifndef  AIRPR
#define AIRPR

struct AIR_PR_data
{
  i_time time;
  int pressure;
};

struct AIR_PR_data get_AIR_PR_data(i_time time);
void init_AIR_PR_file(const char *full_name, i_time time);

#endif
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
/*airpr.c*/
/*-------------------------------------------------------------------------
 * Module:        AIRPR.C
 * Purpose:       Functions associated with reading air pressure file
 * Created:       2/10/91
 *-------------------------------------------------------------------------
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include "globals.h"
#include "datetime.h"
#include "airpr.h"

/* When doing maintenance #define DEBUG here to activate modules
 * auto integrity check.
 */
```

```
/* #define DEBUG */

/*
 * COMMENTARY
 *
 * AIR_PR file format
 *
 * year day_nr time_hr_00 time_hr_12
 *
 * year    = two digit year
 * day     = numer of days from 1/1
 * time_hr_00      = air pressure at 00:00 GMT (in milli bar)
 * time_hr_12      = air pressure at 12:00 GMT
 *
 */

struct AIR_PR_pack
{
  int year;                    /* year of mesurement */
  int day_nr;                  /* number of days from start of year */
  int time_hr_00;  /* air pressure at 00:00 GMT */
  int time_hr_12;  /* air pressure at 12:00 GMT */
};

struct AIR_PR_store
{
  i_time time;
  int time_hr_00;
  int time_hr_12;
};

/*
 * Prototypes for functions local to this module
 */
PRIVATE bool find_AIR_PR_pack(const int year, const int day_nr);
PRIVATE bool read_AIR_PR_pack(struct AIR_PR_pack *pack);
PRIVATE bool find_AIR_PR_store(i_time requested_time);
PRIVATE struct v_time extract_AIR_PR_time(struct AIR_PR_pack *pack);
PRIVATE void AIR_PR_pack_to_store(struct AIR_PR_pack *pack, i_time time,
                                  struct AIR_PR_store *store);
PRIVATE bool read_AIR_PR_store(struct AIR_PR_store *store);
PRIVATE struct AIR_PR_data extract_AIR_PR_data(struct AIR_PR_store *store,
                                               i_time request_time);
PRIVATE struct AIR_PR_store copy_AIR_PR_store(struct AIR_PR_store *store2);

/*
 * Data local to this module
 */
PRIVATE char AIR_PR_name[80];
PRIVATE FILE *AIR_PR_file = NULL;
PRIVATE struct AIR_PR_store store[2];

/*-------------------------------------------------------------------------
 * Low level functions for reading air pressure file
 *-------------------------------------------------------------------------
 */

/*
 * Function:      find_AIR_PR_pack
 * Arguments:     year, day_nr
 * Purpose:       Given year and day_nr find the corresponding entry in the
 *                      in the AIR_PR_file.
 * Returns:             true if successfull else false
 */
bool find_AIR_PR_pack(const int year, const int day_nr)
{
  char buffer[82];
  char *result;
  int search_day;
  int search_year;
  bool found = false;
  long pos;

  assert(AIR_PR_file != NULL);

  while (!feof(AIR_PR_file) && found == false)
  {
    result = fgets(buffer, 80, AIR_PR_file);
    if (result == NULL)
    {
      if (feof(AIR_PR_file))
            return false;
      else
      {
            perror(AIR_PR_name);
            exit(1);
      }
    }

    if (
      sscanf(buffer, "%d %d", &search_year, &search_day)
      == EOF
    )
    {
      fprintf(stderr, "\n%s: %s\n", AIR_PR_name, "Invalid format");
      exit(1);
    }

    if (search_year == year && search_day >= day_nr)
      found = true;
  }

  /* We read past the packet we found ...
   * Reset file to position we had before we read the last line in order
   * to let read_AIR_PR_pack read the packet.
   */
  pos = -((long) strlen(buffer) + 1);
  fseek(AIR_PR_file, pos, SEEK_CUR);

  return found;
}

/*
 * Function:            read_AIR_PR_pack
 * Arguments:           pointer to AIR_PR_pack
 * Purpose:             Find the next entry in the AIR_PR file and put it in the
 *                      given AIR_PR_pack.
 * Returns:             true if sucessfull else false
 */
bool read_AIR_PR_pack(struct AIR_PR_pack *pack)
{
  char buffer[82];
  char *result;

  result = fgets(buffer, 80, AIR_PR_file);
  if (result == NULL)
  {
    if (feof(AIR_PR_file))
      return false;
    else
    {
      perror(AIR_PR_name);
      exit(1);
    }
  }

  if (
    sscanf(buffer, "%d %d %d %d",
      &pack->year,
      &pack->day_nr,
      &pack->time_hr_00,
      &pack->time_hr_12
    ) == EOF
  )
  {
    fprintf(stderr, "\n%s: %s\n", AIR_PR_name, "Invalid format");
    exit(1);
  }

  return true;
}

/*-------------------------------------------------------------------------
 * Air pressure file high level functions
 *-------------------------------------------------------------------------
 */

/*
 * COMMENTARY
 *
 * Why AIR_PR_store ?
 *
 *  As can be seen from the file format there are two entries for each
 * day in the file for the given date.
 *  The POLAR file protocol spesify's that:
 *  1) all requests are in minites since 1/1-1980 00:00 GMT
 *  2) Interface in two functions
 *      the init<>file function sets up reading the files
 *      the read<>data function gets the data with the closest corresponding
 *           time
 *  To accomodate this, two layers are used.
 * One layer reads AIR_PR_store which contains two airpressure entries.
 * get_AIR_PR_data calls read_AIR_PR_data to aquire new objects of this
 * type. Then it passes control to extract_AIR_PR_data wich figures out
 * which of the times to use and uses this in it's AIR_PR_data structure.
 *
 */

/*
 * Function:      find_AIR_PR_store
 * Arguments:     requested_time
 * Purpose:       Find the file entry with the closest date to requested_time.
 * Returns:       true if successful else false.
 */
bool find_AIR_PR_store(i_time requested_time)
{
  char year[5];
  struct v_time time;

  time = minutes_to_v_time(requested_time);

  /* convert four digit year to two digit year */
  sprintf(year    , "%d", time.year);
  sscanf( year + 2, "%d", &time.year);

  return find_AIR_PR_pack(time.year, time.day_nr);
}

/*
 * Function:      extract_AIR_PR_time
 * Arguments:     pointer to AIR_PR_pack
 * Purose:        return a v_time structure containing the
 *                given pack's time info.
 * Returns:       time
 */
struct v_time extract_AIR_PR_time(struct AIR_PR_pack *pack)
{
  struct v_time time;

  time.year   = normalized_year(pack->year); /* four digit year */
  time.day_nr = pack->day_nr;
  time.min_nr = 0;                           /* midnight       */

  return time;
}

/*
 * Function:      AIR_PR_pack_to_store
 * Arguments:     pointer to pack, time, pointer to store
 * Purpose:       convert a AIR_PR_pack structure to a AIR_PR_store struucture
 * Returns:       none
 */
void AIR_PR_pack_to_store(struct AIR_PR_pack *pack, i_time time,
                          struct AIR_PR_store *store)
{
  store->time = time;
  store->time_hr_00 = pack->time_hr_00;
  store->time_hr_12 = pack->time_hr_12;
}

/*
 * Function:      read_AIR_PR_store
 * Arguments:     pointer to AIR_PR_store
 * Purpose:       Read the AIR_PR_store object from the file
 * Returns:       true if sucessful else false
 */
bool read_AIR_PR_store(struct AIR_PR_store *store)
{
  struct AIR_PR_pack pack;
  struct v_time pack_time;
  i_time time;
  bool result;

  result = read_AIR_PR_pack(&pack);
  if (result == true)
  {
   pack_time = extract_AIR_PR_time(&pack);
   time = v_time_to_minutes(&pack_time);
  }

  AIR_PR_pack_to_store(&pack, time, store);

  return result;
}

/*
 * Function:        init_AIR_PR_file
 * Arguments:       full file name, request time
 * Purpose:         setup AIR_PR_file so that all subsequent calls to
 *                  get_AIR_PR_data work.
 * Returns:         none
 */
void init_AIR_PR_file(const char *full_name, i_time time)
{
  strcpy(AIR_PR_name, full_name);

  AIR_PR_file = fopen(AIR_PR_name, "r");
  if (AIR_PR_file == NULL)
  {
   perror(AIR_PR_name);
   exit(1);
  }

  if (find_AIR_PR_store(time) == false)
  {
   fprintf(stderr, "\n%s: %s\n", AIR_PR_name,
            "Unable to find data for that interval");
   exit(1);
  }

  if (
   read_AIR_PR_store(&store[0]) == false ||
   read_AIR_PR_store(&store[1]) == false
  )
  {
   fprintf(stderr, "\n%s: %s\n", AIR_PR_name, "Unexpected EOF");
   exit(1);
  }
}

/*
 * Function:        extract_AIR_PR_data
 * Arguments:       pointer to store object, request time
 * Purpose:         for the request time find out which if the two
 *                  pressure entry's are closer to that time.
 *                  Select this and if nessary 'fix' the time.
 * Returns:         AIR_PR_data object
 */
struct AIR_PR_data extract_AIR_PR_data(struct AIR_PR_store *store,
                                        i_time request_time)
{
  struct u_time time;
  struct v_time total_time;
  struct AIR_PR_data data;
  /* extract time since start of day for request time */
  total_time = minutes_to_v_time(request_time); /* year, day_nr, min_nr */
  time = minutes_to_time(total_time.min_nr);    /* hour, min */

  if (time.hour < 12)
  {
    /* use first interval in store 00:00 - 12:00 */
    data.time = store->time;
    data.pressure = store->time_hr_00;
  }
  else
  {
    /* use second interval in store 12:00 - 24:00 */
    data.time = store->time + 12 * 60; /* add minutes to 12:00 */
    data.pressure = store->time_hr_12;
  }

  return data;
}

/*
 * Function:      copy_AIR_PR_store
 * Arguments:     pointer to AIR_PR_store object
 * Purpose:       copy an AIR_PR_store object.
 * Returns:       copyed object
 */
struct AIR_PR_store copy_AIR_PR_store(struct AIR_PR_store *store2)
{
  struct AIR_PR_store store1;

  store1.time       = store2->time;
  store1.time_hr_00 = store2->time_hr_00;
  store1.time_hr_12 = store2->time_hr_12;

  return store1;
}

/*
 * Function:      get_AIR_PR_data
 * Arguments:     request time
 * Purpose:       Given the request time find the closest corresponding entry
 *                in the file.
 * Returns:       AIR_PR_data object
 */
struct AIR_PR_data get_AIR_PR_data(i_time time)
{
  assert(AIR_PR_file != NULL);

  if (time >= store[1].time)
  {
    do
    {
      /* The second item in the array is better than the first.
       * Make the second item the first and read a new second item.
       */
      store[0] = copy_AIR_PR_store(&store[1]);

      if (read_AIR_PR_store(&store[1]) == false)
      {
          fprintf(stderr, "\n%s: %s\n", AIR_PR_name, "Unexpected EOF");
          exit(1);
      }
    } while (time >= store[1].time);
    /* repeat until this is no longer the case */
  }

  return extract_AIR_PR_data(&store[0], time);
}


/****************************************************************************/

#ifdef DEBUG
void main(void)
{
  struct u_date start_date;
  struct u_date stop_date;
  struct u_time start_time;
  struct u_time stop_time;
  struct v_time cur_time;
  i_time internal_time;
  i_time delta_i;
  i_time termination;
  struct AIR_PR_data data;

  start_date.year = 88; start_date.month = 1; start_date.day = 1;
  start_time.hour = 0;  start_time.min   = 0;

  stop_date.year = 88; stop_date.month = 1; stop_date.day = 2;
  stop_time.hour = 0;  stop_time.min   = 0;

  delta_i = 4;

  puts("Enter date to start sampling AIR_PR data");
  start_date = prompt_date(&start_date);

  puts("Enter time to start sampling AIR_PR data");
  start_time = prompt_time(&start_time);

  puts("Enter date to stop sampling AIR_PR data");
  stop_date = prompt_date(&stop_date);

  puts("Enter time to stop sampling AIR_PR data");
  stop_time = prompt_time(&stop_time);

  puts("Enter sampling frequency");
  scanf("%d", &delta_i);

  internal_time = date_time_to_minutes(&start_date, &start_time);
  termination   = date_time_to_minutes(&stop_date, &stop_time);

  init_AIR_PR_file("C:\\TC\\TCPROG\\DATA\\LTR8788.DAT", internal_time);

  while (internal_time < termination)
  {
   data = get_AIR_PR_data(internal_time);

   cur_time = minutes_to_v_time(internal_time);
   printf("\nRequest time: %2.2d:%3.3d:%4.4d",
     cur_time.year,
     cur_time.day_nr,
     cur_time.min_nr
   );

   cur_time = minutes_to_v_time(data.time);
   printf("\nTime: %2.2d:%3.3d:%4.4d ",
     cur_time.year,
     cur_time.day_nr,
     cur_time.min_nr
   );
   printf("Data: %d\n",
     data.pressure
   );

   internal_time += delta_i;
  }
}
#endif
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
/*uls.h*/
/*
 * header file for ULS.C
 */

#ifndef ULS
#define UIS

struct ULS_data
{
  i_time time;          /* package time converted to system time */
  int  cnt_1;           /* sonar echo counts 1 */
  int  cnt_2;           /* sonar echo counts 2 */
  int  press_transd_cnt; /* pressure transduser count */
  int  angle;                  /* angle of buoy */
  bool approx;          /* true if an approximation  */
};

void init_ULS_file(char *, i_time);
struct ULS_data get_ULS_data(i_time);

#endif
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
/*uls.c*/
/*------------------------------------------------------------------------
 * Module:        ULS.C
 * Purpose:       Functions associated with reading ULS file
 * Created:       30/9/91
 *------------------------------------------------------------------------
 */

#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>
#include "globals.h"
#include "datetime.h"
#include "uls.h"

/* When doing maintenace #define DEBUG here to activate modules
 * auto integrity check.
 */
```

```
/* #define DEBUG */

/*
 * COMMENTARY
 *
 * pack file format
 *
 * The pack files are structures as followed:
 *    yy:ddd:hhmm ssss SSSS pppp a(*)
 *
 * yy    = year's last two digits ec. 1982 = 82
 * ddd   = day number in year ec. 9/21/91 = 264
 * mmmm  = minutes since midnight GMT (Greenwitch Mean Time)
 * ssss  = sonar count 1
 * SSSS  = sonar count 2
 * pppp  = pressure transducer count
 * a(a)  = buoy angle
 * *     = measurement uncertain
 */

struct ULS_pack
{
  int  year;             /* year of measurement */
  int  day_nr;           /* number of day from start of year */
  int  minutes;          /* minutes comensed since midnight */
  int  cnt_1;            /* sonar echo counts 1 */
  int  cnt_2;            /* sonar echo counts 2 */
  int  press_transd_cnt; /* pressure transduser count */
  int  angle;            /* angle of buoy */
  bool approx;           /* true if an approximation  */
};

/*
 * Prototypes for functions local to this module
 */
PRIVATE bool read_ULS_pack(struct ULS_pack *);
PRIVATE bool find_ULS_pack(const int, const int, const int);
PRIVATE char *ULS_file_name(i_time);
PRIVATE bool find_ULS_data(i_time);
PRIVATE struct v_time extract_ULS_time(struct ULS_pack *);
PRIVATE void ULS_pack_to_data(struct ULS_pack *pack, i_time time,
                                    struct ULS_data *data);
PRIVATE bool read_ULS_data(struct ULS_data *);
PRIVATE struct ULS_data copy_ULS_data(struct ULS_data *);


/*
 * Data local to this module
 */
PRIVATE char ULS_name[80];
PRIVATE FILE *ULS_file = NULL;
PRIVATE struct ULS_data data[2];

/*
 * Function:      read_ULS_pack
 * Purpose:       read one line from pack file give file pointer
 * Arguments:     file pointer, pointer to ULS_pack
 * Returns:       true if succeded else false
 */
bool read_ULS_pack(struct ULS_pack *pack)
{
  char buffer[82];
  int pos = 0;
  char *result;

  assert(ULS_file != NULL);

  result = fgets(buffer, 80, ULS_file);
  if (result == NULL)
  {
    return false;
  }

  if (
    sscanf(buffer, "%2d:%3d:%4d %4d %4d %4d %d%n",
      &pack->year,
      &pack->day_nr,
      &pack->minutes,
      &pack->cnt_1,
      &pack->cnt_2,
      &pack->press_transd_cnt,
      &pack->angle,
      &pos
    ) == EOF
  )
  {
    fprintf(stderr, "\n%s: %s\n", ULS_name, "Invalid file format");
    exit(1);
  }

  if (buffer[pos] == '*')
    pack->approx = true;
  else
    pack->approx = false;

  return true;
}

/*
 * Function:       find_ULS_pack
 * Purpose:        Given a time find the corresponding pack.
 *                 Intended to be used in conjunction with read_ULS pack.
 * Arguments:      file pointer, year, day_nr, min_n
 * Returns:        true if successful else false
 */
bool find_ULS_pack(const int year, const int day_nr, const int min_nr)
{
  char buffer[82];
  char *result;
  long pos;
  bool found = false;
  int search_year, search_day_nr, search_min_nr;

  assert(ULS_file != NULL);

  rewind(ULS_file);

  while (!feof(ULS_file) && found == false)
  {
    result = fgets(buffer, 80, ULS_file);
    if (result == NULL)
    {
      if feof(ULS_file)
            return false;
      else
      {
            perror(ULS_name);
            exit(1);
      }
    }

    sscanf(buffer, "%2d:%3d:%4d",
             &search_year, &search_day_nr, &search_min_nr);
    if (normalized_year(search_year) == year && search_day_nr >= day_nr &&
            search_min_nr >= min_nr)
      found = true;
  }

  /* We read past the packet we found ...
   * Reset file to position we had before we read the last line in order
   * to let read_ULS_pack read the packet.
   */
  pos = -((long) strlen(buffer) + 1);
  fseek(ULS_file, pos, SEEK_CUR);

  return found;
}

/*------------------------------------------------------------------------
 * pack file high level functions
 *------------------------------------------------------------------------
 */

/*
 * Function:      ULS_file_name
 * Purpose:       pack file names are always on the form ULSyymm.DAT
 *                yy = year, mm = month
 *                A pack file is always for one month.
 *                Given a date construct an appropriate file name.
 * Arguments:     date
 * Returns:       file name
 */
char *ULS_file_name(i_time time)
{
 struct v_time date_time;
 struct u_date date;
 char name[80];
 char year[5];

 date_time = minutes_to_v_time(time);
 date = day_nr_to_date(date_time.year, date_time.day_nr);

 sprintf(year, "%d", date.year);
 sprintf(name, "ULS%s%2.2d.DAT", (year + 2), date.month);

 return name;
}

/*
 * Function:      find_ULS_data
 * Purpose:       Given system time find closest corresponing data
 *                object.
 * Arguments:     file pointer, requested time
 * Returns:          true if able to find data else false
 */
bool find_ULS_data(i_time requested_time)
{
  struct v_time time;

  time = minutes_to_v_time(requested_time);

  return find_ULS_pack(time.year, time.day_nr, time.min_nr);
}

/*
 * Function:         extract_ULS_time
 * Purpose:          Given the pointer to a ULS_pack exract time
 *                   in v_time format from the packet.
 * Arguments:        pointer to packet
 * Returns:          v_time structure
 */
struct v_time extract_ULS_time(struct ULS_pack *pack)
{
  struct v_time time;

  time.year   = pack->year;
  time.day_nr = pack->day_nr;
  time.min_nr = pack->minutes;

  return time;
}

/*
 * Function:         ULS_pack_to_data
 * Purpose:          Given a pack pack and a system time create a ULS_data object.
 * Arguments:        pack, time
 * Returns:          ULS_data object
 */
void ULS_pack_to_data(struct ULS_pack *pack, i_time time,
                      struct ULS_data *data)
{
  data->time    = time;
  data->cnt_1   = pack->cnt_1;
  data->cnt_2   = pack->cnt_2;
  data->press_transd_cnt = pack->press_transd_cnt;
  data->angle   = pack->angle;
  data->approx  = pack->approx;
}

/*
 * Function:         read_ULS_data
 * Purpose:          Read next ULS_data object from file.
 *                   Almost like read_ULS_pack except i normates time.
 * Arguments:        file pointer, pointer to object to copy data to
 * Returns:          true if able to copy data else false
 */
bool read_ULS_data(struct ULS_data *data)
{
  struct v_time time;
  i_time recieved_time;
  struct ULS_pack pack;
  bool result;

  result = read_ULS_pack(&pack);
  if (result == true)
  {
   time = extract_ULS_time(&pack);
   time.year = normalized_year(time.year);
   recieved_time = v_time_to_minutes(&time);
  }

  ULS_pack_to_data(&pack, recieved_time, data);

  return result;
}

/*
 * Function:       copy_ULS_data
 * Purpose:        copy a ULS_data object
 * Arguments:      pack data pointer
 * Returns:        copyed ULS_data
 */
struct ULS_data copy_ULS_data(struct ULS_data *data2)
{
  struct ULS_data data1;

  data1.time    = data2->time;
  data1.cnt_1   = data2->cnt_1;
  data1.cnt_2   = data2->cnt_2;
  data1.press_transd_cnt  = data2->press_transd_cnt;
  data1.angle   = data2->angle;
  data1.approx  = data2->approx;

  return data1;
}

/*
 * Function:       init_ULS_file
 * Purpose:        called before all other pack functions
 *                 sets up pack high level file system
 * Arguments:      directory name -- the directory in which the pack files reside
 * Returns:        none
 * NOTE:           sets global variables:
 *                   ULS_name
 *                   ULS_data
 *                   ULS_file
 */
void init_ULS_file(char *directory_name, i_time time)
{
  if (ULS_file != NULL) fclose(ULS_file);

  /* ULS_name = directory name + file name */
  strcpy(ULS_name, directory_name);
  strcat(ULS_name, ULS_file_name(time));

  ULS_file = fopen(ULS_name, "r");
  if (ULS_file == NULL)
  {
   perror(ULS_name);
   exit(1);
  }

  if (find_ULS_data(time) == false)
  {
   fputs("\nULS file: Unable to find data for that intervall\n", stderr);
```

```
    exit(1);
  }
  if (
      read_ULS_data(&data[0]) == false ||
      read_ULS_data(&data[1]) == false
  )
  {
    fprintf(stderr,"\n%s :%s\n", ULS_name, "Unexpected EOF");
    exit(1);
  }
}

/*
 * Function:      get_ULS_data
 * Purpose:       given a system time return the data object that most closly
 *                corresponds to this time.
 * Arguments:     time
 * Returns:       ULS_data
 */
struct ULS_data get_ULS_data(i_time time)
{
  assert(ULS_file != NULL);

  if (time >= data[1].time)
  {
    do
    {
      /* The second time in the array in the array is better than the first.
       * Make the second item the first and the read a new second item.
       */
      data[0] = copy_ULS_data(&data[1]);

      if (read_ULS_data(&data[1]) == false)
      {
              /* if unable to read new data try opening a new pack file
               * (next month)
               */
              ULS_name[strlen(ULS_name) - 11] = '\0'; /* remove file name */
              init_ULS_file(ULS_name, time);
      }
    } while (time >= data[1].time);
    /* repeat until this is no longer the case */
  }

  return data[0];
}

/*****************************************************************************/

#ifdef DEBUG
void main(void)
{
  struct u_date start_date;
  struct u_date stop_date;
  struct u_time start_time;
  struct u_time stop_time;
  struct v_time cur_time;
  i_time internal_time;
  i_time delta_i;
  i_time termination;
  struct ULS_data data;
  start_date.year = 88; start_date.month = 1; start_date.day = 1;
  start_time.hour = 0;  start_time.min   = 0;

  stop_date.year = 88; stop_date.month = 1; stop_date.day = 2;
  stop_time.hour = 0;  stop_time.min   = 0;

  delta_i = 4;

  puts("Enter date to start sampling ULS data");
  start_date = prompt_date(&start_date);

  puts("Enter time to start sampling ULS data");
  start_time = prompt_time(&start_time);

  puts("Enter date to stop sampling ULS data");
  stop_date = prompt_date(&stop_date);

  puts("Enter time to stop sampling ULS data");
  stop_time = prompt_time(&stop_time);

  puts("Enter sampling frequency");
  scanf("%d", &delta_i);

  internal_time = date_time_to_minutes(&start_date, &start_time);
  termination   = date_time_to_minutes(&stop_date, &stop_time);

  init_ULS_file("C:\\TC\\TCPROG\\DATA\\", internal_time);

  while (internal_time < termination)
  {
   data = get_ULS_data(internal_time);

   cur_time = minutes_to_v_time(internal_time);
   printf("\nRequest time: %2.2d:%3.3d:%4.4d",
     cur_time.year,
     cur_time.day_nr,
     cur_time.min_nr
   );

   cur_time = minutes_to_v_time(data.time);
   printf("\nTime: %2.2d:%3.3d:%4.4d ",
     cur_time.year,
     cur_time.day_nr,
     cur_time.min_nr
   );
   printf("Data: %4.4d %4.4d %4.4d %d ",
     data.cnt_1,
     data.cnt_2,
     data.press_transd_cnt,
     data.angle
   );
   if (data.approx == true)
     printf("*\n");
   else
     printf("\n");

   internal_time += delta_i;
  }
}
#endif
```

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
/*draftout.h*/
/*
 * header file for DRAFTOUT.C
 */

#ifndef DRAFTOUT
#define DRAFTOUT

struct EXTRA_data
{
  i_time start_time;
  i_time stop_time;
  double longtitude;
  double latitude;
  double lower_limit;
  int fall_out;
};

struct DRAFT_data
{
  i_time time;
  double draft;
  bool approx;
  double debth;
};

struct STAT_data
{
  int samples;
  double min_value;
  double max_value;
  double average;
  double mean_deviation;
  double standard_deviation;
  double varians;
  bool   no_varians;
  double skew;
  double curtosis;
};

void init_DRAFT_file(void);
void put_DRAFT_data(struct DRAFT_data *);

void init_STAT_file(void);
void put_STAT_data(struct STAT_data *, struct EXTRA_data *);

#endif
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
/*draftout.c*/
/*-------------------------------------------------------------------------
 * Module:       draftout.c
 * Purpose:      handle the output files for draft
 *               That is a file for statistics and a more general log file
 * Created:      3/10/91
 *-------------------------------------------------------------------------
 */

#include <stdlib.h>
```

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <time.h>
#include <assert.h>
#include "globals.h"
#include "validate.h"
#include "datetime.h"
#include "prompt.h"
#include "draftout.h"

/*
 * structures local to this module
 */
struct DRAFT_information
{
  bool header;
  bool time;
  bool draft;
  bool approx;
  bool debth;
};

/*
 * Prototypes local to this module
 */
PRIVATE void prompt_DRAFT_info(void);
PRIVATE void write_DRAFT_header(void);

/*
 * Data local to this module
 */
PRIVATE char DRAFT_name[80];
PRIVATE FILE *DRAFT_file = NULL;
PRIVATE struct DRAFT_information DRAFT_info =
  {
    false, /* header  */
    false, /* time    */
    false, /* draft   */
    false, /* approx  */
    false /* debth    */
  };

PRIVATE char STAT_name[80];
PRIVATE FILE *STAT_file = NULL;

/*-------------------------------------------------------------------------
 * DRAFT log file
 *-------------------------------------------------------------------------
 */

/*
 * Function:      prompt_DRAFT_info
 * Arguments:     none
 * Purpose:       Present a menu of all possible file options and make the
 *                user select the items he wants on the file.
 * Returns:       none
 */
void prompt_DRAFT_info(void)
{
  const char menu[] =
 "Select the information you want in the table file:" "\n\n"
 "\t" "A : Information header                " "\n"
 "\t" "B : Date information (year:day:minutes)" "\n"
  "\t" "C : Draft Information                 " "\n"
  "\t" "D : Aproximaton marks                 " "\n"
  "\t" "E : ULS depth                         " "\n";

const char menu_response[] =
  "?~A, ?~B, ?~C, ?~D, ?~E";

static char buffer[80];
char *string;
bool OK;

puts(menu);

do
{
  printf("Menu (Format: (a)(b)(c)(d)(e)) (Default: abcde) -> ");

  fgets(buffer, 80, stdin);
  string = frontstrip(buffer);

  if (strlen(string) == 0 || *string == '\n')
  {
   /* accept default */
   string = strcpy(buffer, "abcde");
   OK = true;
  }
  else
   OK = validate(menu_response, string);
} while (OK == false);

while(strlen(string) != 0 && *string != '\n')
{
  switch (toupper(*string))
  {
   case 'A': DRAFT_info.header  = true; break;
   case 'B': DRAFT_info.time    = true; break;
   case 'C': DRAFT_info.draft   = true; break;
   case 'D': DRAFT_info.approx  = true; break;
   case 'E': DRAFT_info.debth   = true; break;
   default:
       fputs("Function prompt_out_info: Internal error", stderr);
       exit(1);
}

  string++;
}
}

/*--------------------------------------------------------------------------
* DRAFT log file
*--------------------------------------------------------------------------
*/

/*
* Function:     write_DRAFT_header
* Arguments:    none
* Purpose:      Write the file header if one was requested.
*               Write what information is stored on the file.
* Returns:          none:
*/
void write_DRAFT_header(void)
{
bool first = true;
time_t c_time;

assert(DRAFT_file != NULL);

if (DRAFT_info.header == true)
{
  if (
  fputs("File log for draft computaion", DRAFT_file) == EOF
  )
  {
   perror(DRAFT_name);
   exit(1);
  }

  c_time = time(NULL);
  if (
  fprintf(DRAFT_file, "\nCreated: %s", asctime(localtime(&c_time))) == EOF
  )
  {
   perror(DRAFT_name);
   exit(1);
  }

  if (
  fputs("\nInformation on file is:\n", DRAFT_file) == EOF
  )
  {
   perror(DRAFT_name);
   exit(1);
  }

  if (DRAFT_info.time    == true)
  {
   if (
   fputs("Time", DRAFT_file) == EOF
   )
   {
        perror(DRAFT_name);
        exit(1);
   }
   first = false;
  }
  if (DRAFT_info.draft   == true)
  {
   if (first == false)
   {
        if (
        fputs(",\t", DRAFT_file) == EOF
        )
        {
         perror(DRAFT_name);
         exit(1);
        }
   }
   else
        first = false;
 if (
 fputs("Draft", DRAFT_file) == EOF
 )
 {
        perror(DRAFT_name);
        exit(1);
 }
}
if (DRAFT_info.approx  == true)
{
 if (first == false)
 {
        if (
        fputs(",\t", DRAFT_file) == EOF
        )
        {
          perror(DRAFT_name);
          exit(1);
        }
 }
 else
        first = false;
 if (
 fputs("Approx. mark", DRAFT_file) == EOF
 )
 {
        perror(DRAFT_name);
        exit(1);
 }
}
if (DRAFT_info.debth   ==true)
{
 if (first == false)
 {
        if (
        fputs(",\t", DRAFT_file) == EOF
        )
        {
          perror(DRAFT_name);
          exit(1);
        }
 }
 else
        first = false;
 if (
 fputs("Sonar Depth", DRAFT_file) == EOF
 )
 {
        perror(DRAFT_name);
        exit(1);
 }
}
if (
fputs("\n#", DRAFT_file) == EOF
)
{
 perror(DRAFT_name);
 exit(1);
}
if (
fputs("\n", DRAFT_file) == EOF
```

```
    )
    {
      perror(DRAFT_name);
      exit(1);
    }
  }
}

/*
 * Function:      put_DRAFT_data
 * Arguments:     pointer to DRAFT_data
 * Purpose:       Write the requested information from the DRAFT_data
 *                structure to the file.
 * Returns:       none
 */
void put_DRAFT_data(struct DRAFT_data *data)
{
  bool first = true;
  struct v_time time;
  char year[5];

  if (DRAFT_file == NULL) return;

  if (DRAFT_info.time    == true)
  {
    time = minutes_to_v_time(data->time);
    sprintf(year, "%d", time.year);
    sscanf(year+2, "%d", &time.year);
    if (
    fprintf(DRAFT_file, "%2.2d:%3.3d:%4.4d",
              time.year, time.day_nr, time.min_nr) == EOF
    )
    {
      perror(DRAFT_name);
      exit(1);
    }
    first = false;
  }

  if (DRAFT_info.draft   == true)
  {
    if (first == false)
    {
      if (
      fputs("\t", DRAFT_file) == EOF
      )
      {
          perror(DRAFT_name);
          exit(1);
      }
    }
    else
      first = false;

    if (
    fprintf(DRAFT_file, "%7.3lf", data->draft) == EOF
    )
    {
      perror(DRAFT_name);
      exit(1);
    }
```

```
}

if (DRAFT_info.approx  ==true)
{
 if (first == false)
 {
  if (
  fputs("\t", DRAFT_file) == EOF
  )
  {
        perror(DRAFT_name);
        exit(1);
  }
 }
 else
  first = false;

 if (data->approx ==true)
 {
  if (
  fprintf(DRAFT_file, "  *") == EOF
  )
  {
        perror(DRAFT_name);
        exit(1);
  }
 }
 else
 {
  if (
  fprintf(DRAFT_file, "   ") == EOF
  )
  {
        perror(DRAFT_name);
        exit(1);
  }
 }
}

if (DRAFT_info.debth   == true)
{
 if (first == false)
 {
  if (
  fputs("\t", DRAFT_file) == EOF
  )
  {
        perror(DRAFT_name);
        exit(1);
  }
 }
 else
  first = false;

 if (
 fprintf(DRAFT_file, "%8.5lf", data->debth) == EOF
 )
 {
  perror(DRAFT_name);
  exit(1);
 }
```

```
  }

  if (first == false)
  {
    if (
    fputs("\n", DRAFT_file) == EOF
    )
    {
      perror(DRAFT_name);
      exit(1);
    }
  }
}

/*
 * Function:        init_DRAFT_file
 * Arguments:       none
 * Purpose:         1) prompt if the user wants a draft file
 *                  2) if so get the file name
 *                  3) open the file
 *                  4) get the information the user wishes to put in the file
 * Returns:         none
 * NOTE:            must be called prior to any call to put_DRAFT_data
 */
void init_DRAFT_file(void)
{
  char *file_name;

  puts("");
  puts("Do you want a draft table file ?");
  if (prompt_bool(false) == true)
  {
    puts("");
    puts("Enter draft table file name");
    file_name = prompt_file("C:\\TC\\TCPROG\\TABLE.TBL");

    strcpy(DRAFT_name, file_name);

    DRAFT_file = fopen(file_name, "w");
    if (DRAFT_file ==  NULL)
    {
      perror(file_name);
      exit(1);
    }

    puts("");
    prompt_DRAFT_info();

    write_DRAFT_header();
  }
}

/*-------------------------------------------------------------------------
 * STAT log file
 *-------------------------------------------------------------------------
 */

void put_STAT_data(struct STAT_data *data, struct EXTRA_data *extra)
{
  /*
   * Prints:
```

```
* Time printed
* Start time of sampling
* Stop time of sampling
* Latitude
* Longtitude
* Number of aproximate values
* Average draft
* Number of valid samples
* Number of samples out of limits
* Maximum value
* Minimum value
* Lower limit
* Mean deviation
* Standard deviation
* Varians
* Skew
* Curtosis
*/
time_t c_time;
struct v_time v_start, v_stop;
struct u_date start_date, stop_date;
struct u_time start_time, stop_time;

if (STAT_file == NULL) return;

fprintf(STAT_file, "STATISTICS FILE\n");
fprintf(STAT_file, "===============\n");
fprintf(STAT_file, "\n");

if (ferror(STAT_file))
{
  perror(STAT_name);
  exit(1);
}

/* get current time and print it */
c_time = time(NULL);
fprintf(STAT_file, "Time printed: %s\n", asctime(localtime(&c_time)));

fprintf(STAT_file, "\n");
if (ferror(STAT_file))
{
  perror(STAT_name);
  exit(1);
}

/* Convert internal time (minutes since 1/1-1980 00:00 GMT)
 * to standard english time and date format (mm/dd-yyyy hh:mm).
 * Do this for both start and stop time and print them.
 */
v_start = minutes_to_v_time(extra->start_time);
start_date = day_nr_to_date(v_start.year, v_start.day_nr);
start_time = minutes_to_time(v_start.min_nr);

v_stop = minutes_to_v_time(extra->stop_time);
stop_date = day_nr_to_date(v_stop.year, v_stop.day_nr);
stop_time = minutes_to_time(v_stop.min_nr);

fprintf(STAT_file, "Sampling start time: %2d/%2d-%4d %2.2d:%2.2d GMT\n",
  start_date.month, start_date.day, start_date.year,
  start_time.hour, start_time.min
 );
fprintf(STAT_file, "Sampling  stop time: %2d/%2d-%4d %2.2d:%2.2d GMT\n",
  stop_date.month, stop_date.day, stop_date.year,
  stop_time.hour, stop_time.min
 );
fprintf(STAT_file, "\n");

 fprintf(STAT_file, "\n");

 if (ferror(STAT_file))
 {
  perror(STAT_name);
  exit(1);
 }

 /* print the rest of the generics in the extra struct */
 fprintf(STAT_file, "Lower accepted limit: %8.4lf\n", extra->lower_limit);
fprintf(STAT_file, "Number of values not inside limit: %5d\n", extra->fall_out);
fprintf(STAT_file, "\n");

fprintf(STAT_file, "\n");
 if (ferror(STAT_file))
{
  perror(STAT_name);
  exit(1);
}

/* print the statistical data */
fprintf(STAT_file, "Number of samples:    %5d\n", data->samples);
fprintf(STAT_file, "Average draft:        %8.4lf\n", data->average);
fprintf(STAT_file, "Minimum value:        %8.4lf\n", data->min_value);
fprintf(STAT_file, "Maximum value:        %8.4lf\n", data->max_value);
fprintf(STAT_file, "Mean deviation:       %8.4lf\n", data->mean_deviation);
fprintf(STAT_file, "Standard deviation: %8.4lf\n", data->standard_deviation);
fprintf(STAT_file, "Varians:              %8.4lf\n", data->varians);
fprintf(STAT_file, "\n");
 if (ferror(STAT_file))
{
  perror(STAT_name);
  exit(1);
}

/* if skew and kurtosis have been computed (varians <> 0) print them */
 if (data->no_varians == false)
{
  fprintf(STAT_file, "Skew:     %8.4lf\nKurtosis: %8.4lf",
    data->skew,
    data->curtosis
  );
}
else
{
  fprintf(STAT_file, "Varians = 0 ==> no skew or kurtosis");
}
fprintf(STAT_file, "\n");

if (ferror(STAT_file))
{
  perror(STAT_name);
  exit(1);
}
```

```
}

void init_STAT_file(void)
{
 char *name;

 puts("Do you want the statistics printed to a file ?");
 if (prompt_bool(true))
 {
  puts("");
  puts("Enter file name:");
  /*vvvvvvvvvvvvvvvvvv   path maybe to be changed   vvvvvvvvvvvvvvvvvvv*/
  name = prompt_file("C:\\TC\\TCPROG\\STAT.TBL");
  strcpy(STAT_name, name);

  STAT_file = fopen(STAT_name, "w");
  if (STAT_file == NULL)
  {
   perror(STAT_name);
   exit(1);
  }
 }
}
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
/*drift.h*/
/*
 * Include file for draft.c
 */

struct DRIFT_data
{
 bool  have_drift;
 double per_day;
 double total;
};

struct DRIFT_data drift(void);
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
/*drift.c*/
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <string.h>
#include <assert.h>
#include "globals.h"
#include "prompt.h"
#include "drift.h"

/*Nr of header lines, not to be read. */
#define NR_OF_NP_JUMPLINES 4
/* Nr of header lines, not to be read. */
#define NR_OF_ZH_JUMPLINES 2
/* Sampling each fourth minute.*/
#define NR_OF_SAMP_A_DAY 360
#define NR_OF_SAMPLES 360
/*Degrees / radian */
#define DEG_TO_RAD 57.29577951
/*International def. */
#define NAUT_MILE_TO_KM 1.852
```

```
/* [m] WGS84*/
#define HALF_MINOR_AXIS 6378137.0
/*WGS84*/
#define EXCENTR 0.00669438

#define CORR_DUMMY 40 /*Dummy to correct the frameradii. To be removed if the
                        scale error in radius1 and radius2 is found.
                        (in sort_vect_out_sum).*/

/************************************************************************/

PRIVATE
double xframedist[6],
     yframedist[6],
     frameweight[6],
     cframedist[6];

PRIVATE
double frame_form,
     nr_of_frames;

PRIVATE
int aproxnr,
   s_nr,
   vect_nr;

PRIVATE
long data_file_type,
    stop_sign;

PRIVATE
double centr_utmlat,
     centr_utmlon,
     res_vect,
     azimut,
     centrlat,
     centrlon;

PRIVATE
double st_d_lon,
     st_d_lat,
     st_d_x,
     st_d_y;

PRIVATE
double sum_p_dlambda_cosphi,
     sum_p_meanphi,
     sum_p_meanlambda,
     sum_p_dphi,
     sum_p_dx,
     sum_p_dy,
     sum_p_xsqr,
     sum_p_ysqr,
     sum_p_phisqr,
     sum_p_lambdasqr,
     sum_p;

PRIVATE
double sum_sqrp_sqrphi,
     sum_sqrp_sqrlambda,
     sum_sqrp_sqrx,
    sum_sqrp_sqry;

struct latlon{
   double lat;
   double lon;
   };

PRIVATE
struct latlon corner1[6],
              corner2[6],
              corner3[6],
              corner4[6];

PRIVATE
struct latlon startcoor,
              endcoor;

PRIVATE
char drift_file_name[80];

PRIVATE
FILE *drift_fp = NULL;

/**************************************************************************/

/*
 * Prototypes
 */
PRIVATE int get_drift_file(void);
PRIVATE void read_first_on_file(void);
PRIVATE void read_coor(void);
PRIVATE void ask_frame(void);
PRIVATE void frame_in(void);

PRIVATE void gtu(double *x, double *y, double sm);
PRIVATE double mean_curvature(double phi);
/* PRIVATE double perp_curv(double phi); */
PRIVATE void sum_init(void);
PRIVATE void sum_dl_dp(double weight, int *summed);
PRIVATE void sum_dx_dy(double weight, int *summed);
PRIVATE void sort_vect_out_sum(void);
PRIVATE void get_mean_xy(void);
PRIVATE void get_mean_lp(void);
PRIVATE void set_rect_corners(int i);

/**************************************************************************/

int get_drift_file(void)
{
  char *name;

  puts("");
  puts("Do you have any drift vector file?");

  if (prompt_bool(true) == true)
  {
    puts("");
    puts("Give complete filename of the drift vector file.");
    name = prompt_file("C:\\TC\\TCPROG\\DATA\\Z23.DAT");
    strcpy(drift_file_name, name);
    puts("");
    printf("\n %s\t %s\n\t\t\t %s\n\t\t\t %s\n\n",
      "Give file format", "1. ARC/INFO-format",
                                    "2. Zhang-format",
                                    "3. NP-format"
    );
    do
    {
      data_file_type = prompt_integer(2);
    } while (data_file_type < 1 && data_file_type > 3);

    if ((drift_fp = fopen(drift_file_name, "r")) == NULL)
    {
      perror(drift_file_name);
      exit(1);
    }
    else
    {
      return (1);
    }
  }

  return (0);
} /*get_drift_file..*/

/***************************************************************************/

/*To read to the beginning of position data
 */
void read_first_on_file(void)
{
  int i;
  char buffer[256];

  assert(drift_fp != NULL);

  switch (data_file_type)
    {
    case (1):
      break;

    case (2):
      for(i=0;i<=NR_OF_ZH_JUMPLINES;i++)
              fgets(buffer, 256, drift_fp);
      break;

    case (3):
      for(i=0;i<=NR_OF_NP_JUMPLINES;i++)
              fgets(buffer, 256, drift_fp);
      break;

    default:
      break;
    }
}

/***************************************************************************/

void read_coor(void)
{
  char buffer[256];
char typec;
stop_sign = 0;

assert(drift_fp != NULL);

switch(data_file_type)
 {
   case 1:
            fgets(buffer, 256, drift_fp);
            if (ferror(drift_fp))
            {
              perror(drift_file_name);
              exit(1);
            }

            fgets(buffer, 256, drift_fp);
            if (ferror(drift_fp))
            {
              perror(drift_file_name);
              exit(1);
            }

            if (
            sscanf(buffer, "%lf %lf", &(startcoor.lat), &(startcoor.lon))
              == EOF
            )
            {
              fprintf(stderr, "\n%s: %s\n", drift_file_name,
                "Invalid file format");
              exit(1);
            }

            fgets(buffer, 256, drift_fp);
            if (ferror(drift_fp))
            {
              perror(drift_file_name);
              exit(1);
            }

            if (
            sscanf(buffer, "%lf %lf", &(endcoor.lat), &(endcoor.lon))
            == EOF
            )
            {
              fprintf(stderr, "\n%s: %s\n", drift_file_name,
                "Invalid file format");
              exit(1);
            }

            fgets(buffer, 256, drift_fp);
            if (ferror(drift_fp))
            {
              perror(drift_file_name);
              exit(1);
            }
            break;

    case 2:
            fgets(buffer, 256, drift_fp);
            if (ferror(drift_fp))
            {
```

```
            perror(drift_file_name);
            exit(1);
           }

           if (
           sscanf(buffer, " %c%*d %lf %lf %lf %lf",
             &typec,
             &(startcoor.lat), &(startcoor.lon),
             &(endcoor.lat),  &(endcoor.lon)
           ) != 5
           )
           {
            fprintf(stderr, "\n%s: %s\n", drift_file_name,
              "Invalid file format");
            exit(1);
           }

           if(typec == 'E')
            stop_sign = 1;
           break;

    case 3:
           fgets(buffer, 256, drift_fp);
           if (ferror(drift_fp))
           {
            perror(drift_file_name);
            exit(1);
           }

           if (
           sscanf(buffer, "%*s %lE %lE",
             &(startcoor.lat), &(endcoor.lon)
           ) == EOF
           )
           {
            fprintf(stderr, "\n%s: %s\n", drift_file_name,
              "Invalid file format");
            exit(1);
           }
           break;

    default:
           fputs("\nInternal error: function read_coord\n", stderr);
           exit(1);
  }
}

/****************************************************************************/

void ask_frame(void)
{
 puts("");
 printf("%s\n\n","Give prefered frame form for the vector weighting.");
 printf("\t\t%s\n","1. Concentric.");
 printf("\t\t%s\n","2. Rectangular.");
 puts("");

 do
 {
  frame_form = prompt_integer(2);
  } while (frame_form != 2 && frame_form != 1);

}

/**************************************************************************/

void frame_in(void)
{
  int i;
  double f1, f2;

  puts("");
  puts("Give the center latitud.");

  do
  {
    centrlat = prompt_real(76.0);
  } while (centrlat < -180.0 && centrlat > 180.0);

  puts("");
  puts("Give the center longitude.");

  do
  {
    centrlon = prompt_real(-20.0);
  } while (centrlon < -180.0 && centrlon > 180.0);

  puts("");
  puts("How many frames do you want for the drift vector computation? (1-5)."
    );

  do
  {
    nr_of_frames = prompt_integer(1);
  } while ( nr_of_frames < 1 && nr_of_frames > 5);

  switch(frame_form)
  {
    case (1):
              for(i=0; i<=nr_of_frames-1;i++)
              {
                printf(
                  "\nGive the radius from the center for frame %d (in km).\n",
                  i+1
                );
                f1 = prompt_real(100.0);

                cframedist[i] = f1;
              }
              break;

    case (2):
      for(i=0; i<=nr_of_frames-1;i++)
            {
              printf(
                "\nGive the N-S extension from the center for frame %d (in km).\n",
                i+1
              );
              f1 = prompt_real(100.0);

              printf(
            "\nGive the E_W extension from the center for frame %d (in km).\n",
            i+1
           );
           f2 = prompt_real(100.0);

           xframedist[i] = f1;
           yframedist[i] = f2;
         }
      break;
  }

  for(i=0; i<=nr_of_frames-1;i++)
  {
    printf("\nGive the weight for frame %d.\n",i+1);
    f1 = prompt_real(1.0);

    frameweight[i] = f1;
  }
}

/*****************************************************************************/

void gtu(double *x, double *y, double sm)
{
  static double ro  = 57.2957795;
  static double ra  = 6378388.00;
  static double e2  = 0.00672267;
  static double e2m = 0.00676817;
  static double f1a = 6365107.43;
  static double f1b = 16100.5912;
  static double f1c = 16.9694;
  static double f2a = 0.001174751;
  static double f3a = 0.00000023;
  static double f3b = 0.06091353;
  static double f3c = 0.000183233;
  static double f4a = 0.048461975;
  static double f5a = 0.000018985;

  static double fi,p,p2,sf,cf,cf2,tf,tf2,rn,cn,scn,f1,f2,f3,f4,f5;
  static int ns;

  ns = 0;
  if( *y < 0.0 ) {
    ns = -1;
    *y = -(*y);
  }
  fi  = (*y)/ro;
  p   = ( (*x) - (sm) )*0.36;
  p2  = p*p;
  sf  = sin( fi );
  cf  = cos( fi );
  cf2 = cf * cf;
  tf  = sf/cf;
  tf2 = tf*tf;
  rn  = ra/sqrt(1.0 - e2*sf*sf);
  cn  = rn*cf;
  scn = cn*sf;
  f1  = f1a*fi - f1b*sin( fi*2.0 ) + f1c*sin( fi*4.0 );
  f2  = f2a*scn;
  f3  = f3a*scn*cf2*( 5.0-tf2+cf2*(f3b+f3c*cf2));
  f4  = f4a*cn;
```

```
 f5  = f5a*cn*cf2*( 1.0-tf2+e2m*cf2);

 *y  = f1+p2*(f2+f3*p2);
 *x  = 500000.0 + p*(f4+f5*p2);
 if(ns < 0 ) *y = 10000000.0 - *y;
}

/***************************************************************************/

double mean_curvature(double phi)
{
 double W, radius;

 W = sqrt(1- EXCENTR*EXCENTR*sin(phi/DEG_TO_RAD)*sin(phi/DEG_TO_RAD));
 radius = HALF_MINOR_AXIS*(sqrt(1- EXCENTR*EXCENTR))/W*W;

 return (radius);
}

/***************************************************************************/
/*
double perp_curv(double phi)
{
 double percur;

 percur = HALF_MINOR_AXIS /
   sqrt(
     (1.0 - EXCENTR*EXCENTR * sin(phi/DEG_TO_RAD) * sin(phi/DEG_TO_RAD))
   );

 return (percur);
}
*/
/***************************************************************************/

void sum_init(void)
{
 sum_p_dlambda_cosphi = 0.0;
 sum_p_meanphi = 0.0;
 sum_p_meanlambda = 0.0;
 sum_p_dphi = 0.0;
 sum_p_phisqr = 0.0;
 sum_p_lambdasqr = 0.0;
 sum_sqrp_sqrphi = 0.0;
 sum_sqrp_sqrlambda = 0.0;
 sum_p_dx = 0.0;
 sum_p_dy = 0.0;
 sum_p_xsqr = 0.0;
 sum_p_ysqr = 0.0;
 sum_sqrp_sqrx = 0.0;
 sum_sqrp_sqry = 0.0;
 sum_p = 0.0;
}

/***************************************************************************/

void sum_dl_dp(double weight, int *summed)
{
 double large_lambda, small_lambda,
          large_phi, small_phi;
 double delta_lambda, delta_phi,
            mean_lambda, mean_phi;

  startcoor.lon /= DEG_TO_RAD;
  startcoor.lat /= DEG_TO_RAD;
  endcoor.lon   /= DEG_TO_RAD;
  endcoor.lat   /= DEG_TO_RAD;

  large_lambda = (startcoor.lon >= endcoor.lon) ? startcoor.lon : endcoor.lon;
  small_lambda = (startcoor.lon < endcoor.lon)  ? startcoor.lon : endcoor.lon;
  large_phi    = (startcoor.lat >= endcoor.lat) ? startcoor.lat : endcoor.lat;
  small_phi    = (startcoor.lat < endcoor.lat)  ? startcoor.lat : endcoor.lat;

  delta_lambda = large_lambda - small_lambda;
  delta_phi = large_phi - small_phi;

  mean_lambda = delta_lambda/2.0 + small_lambda;
  mean_phi = delta_phi/2.0 + small_phi;

  sum_p                 += weight;
  sum_p_dlambda_cosphi += weight*delta_lambda*cos(mean_phi);
  sum_p_meanphi         += weight*mean_phi;
  sum_p_meanlambda      += weight*mean_lambda;
  sum_p_dphi            += weight*delta_phi;
  sum_p_lambdasqr       += weight*delta_lambda;
  sum_p_phisqr          += weight*delta_phi*delta_phi;
  sum_sqrp_sqrphi       += weight*weight*delta_phi*delta_phi;
  sum_sqrp_sqrlambda    += weight*weight*delta_lambda*delta_lambda;

  *summed = 1;
}

/****************************************************************************/

void sum_dx_dy(double weight, int *summed)
{
  double deltax, deltay;

  deltax = fabs(startcoor.lat - endcoor.lat);
  deltay = fabs(startcoor.lon - endcoor.lon);

  sum_p    += weight;

  sum_p_dx      += weight * deltax;
  sum_p_dy      += weight * deltay;

  sum_p_xsqr    += weight * deltax * deltax;
  sum_p_ysqr    += weight * deltay * deltay;

  sum_sqrp_sqrx += weight * weight * deltax * deltax;
  sum_sqrp_sqry += weight * weight * deltay*deltay;

  *summed = 1;
}

/****************************************************************************/

void sort_vect_out_sum(void)
{
  double lat1, lat2, lat3,
            lon1, lon2, lon3;
```

```
double s1, s2,
        radius1, radius2;

int i, summed;

switch(frame_form)
{
 case (1):
  lat1 = startcoor.lat/DEG_TO_RAD;
  lat2 = centrlat/DEG_TO_RAD;
  lat3 = endcoor.lat/DEG_TO_RAD;
  lon1 = startcoor.lon/DEG_TO_RAD;
  lon2 = centrlon/DEG_TO_RAD;
  lon3 = endcoor.lon/DEG_TO_RAD;

  s1 = sin(lat1)*sin(lat2) + cos(lat1)*cos(lat2)*cos(lon1-lon2);
  s2 = sin(lat3)*sin(lat2) + cos(lat3)*cos(lat2)*cos(lon3-lon2);

  radius1 = 60.00*acos(s1)*NAUT_MILE_TO_KM*CORR_DUMMY;
  radius2 = 60.00*acos(s2)*NAUT_MILE_TO_KM*CORR_DUMMY;

  summed = 0;
  for(i=0;i<nr_of_frames;i++)
       {
        if(summed == 0)
         if((radius1 < cframedist[i]) || (radius2 < cframedist[i]))
          {
          sum_dl_dp(frameweight[i], &summed);
          vect_nr++;
         }
       }
  break;

 case (2):
  gtu(&(startcoor.lon), &(startcoor.lat), centrlon);
  gtu(&(endcoor.lon), &(endcoor.lat), centrlon);

  /*x-direction to the east in gtu. */
  summed = 0;

  for(i=0;i<nr_of_frames;i++)
       {
        if(summed == 0)
         {
         if(((startcoor.lat <= corner1[i].lat) &&
                (startcoor.lat >= corner4[i].lat)) &&
               ((startcoor.lon <= corner2[i].lon) &&
                (startcoor.lon >= corner3[i].lon)))
          {
          if(((endcoor.lat <= corner1[i].lat) &&
                 (endcoor.lat >= corner4[i].lat)) &&
                 ((endcoor.lon <= corner2[i].lon) &&
                  (endcoor.lon >= corner3[i].lon)))
                {
                sum_dx_dy(frameweight[i], &summed);
                vect_nr++;
                }
          }
         }
       }
  break;
 }
}

/**************************************************************************/

void get_mean_xy(void)
{
 double meanx, meany, angle;

 if (sum_p != 0.0)
 {
  meanx = sum_p_dx/sum_p; /*x-component*/
  meany = sum_p_dy/sum_p; /*y_component*/
 }
 else
  meanx = meany = 0.0;

 res_vect = sqrt(meanx*meanx + meany*meany);
 angle = atan2(meany, meanx);              /*Math. coord. vs map coord. */
 azimut = angle*DEG_TO_RAD + 180;

 if (sum_p != 0)
 {
  st_d_x = sum_p_xsqr - sum_sqrp_sqrx/sum_p;
  st_d_y = sum_p_ysqr - sum_sqrp_sqry/sum_p;
 }
 else
  st_d_x = st_d_y = 0.0;
}

/**************************************************************************/

void get_mean_lp(void)
{
 double tielat, /*tielon, */
          meanlat, meanlon,
          res_vect_rad, angle_rad,
          meanR;

 /* OBS OBS OBS!!!!All sums are in radians (exept sum_p) from sum_dl_dp.*/
 tielat = sum_p_meanphi/sum_p;
 /*tielon = sum_p_meanlambda/sum_p;*/

 if (sum_p != 0.0)
 {
  meanlon = sum_p_dlambda_cosphi/(cos(tielat)*sum_p);
  meanlat = sum_p_dphi/sum_p;
 }
 else
  meanlon = meanlat = 0.0;

 res_vect_rad = acos(cos(meanlat)*cos(meanlon));
 angle_rad = acos(
                  ( cos(meanlon) - cos(res_vect_rad) * cos(meanlat) ) /
                  ( sin(res_vect_rad) * sin(meanlat) )
              );

 if (sum_p != 0.0)
 {
  st_d_lat = sum_p_phisqr    - sum_sqrp_sqrphi    / sum_p;
  st_d_lon = sum_p_lambdasqr - sum_sqrp_sqrlambda / sum_p;
 }
 else
   st_d_lat = st_d_lon = 0.0;

 st_d_lat *= DEG_TO_RAD; /*Here radians to degrees. */
 st_d_lon *= DEG_TO_RAD; /*Here radians to degrees. */

 meanR = mean_curvature(tielat);

 res_vect = res_vect_rad*meanR;
 azimut = angle_rad*DEG_TO_RAD + 180; /*Here radians to degrees */
}

/**************************************************************************/

void set_rect_corners(int i)
{
 centr_utmlat = centrlat;
 centr_utmlon = centrlon;

 gtu(&centr_utmlon, &centr_utmlat, centrlon);

 corner1[i].lat = corner2[i].lat = centr_utmlat + xframedist[i]*1000;
 corner3[i].lat = corner4[i].lat = centr_utmlat - xframedist[i]*1000;
 corner1[i].lon = corner3[i].lon = centr_utmlon - yframedist[i]*1000;
 corner2[i].lon = corner4[i].lon = centr_utmlon + yframedist[i]*1000;
}

/**************************************************************************/

struct DRIFT_data drift(void)
{
 int i;
 struct DRIFT_data data;
 double per_sample;

 res_vect = 0.0;
 azimut = -1.0;
 vect_nr = 0;
 aproxnr = 0;
 s_nr = 0;
 nr_of_frames = 0;

 data.have_drift = false;
 data.per_day = data.total = 0.0;

 sum_init();
 if(get_drift_file())
 {
   ask_frame();
   frame_in();

   if(frame_form == 2)
   {
    for(i=0;i<=nr_of_frames-1;i++)
           set_rect_corners(i);
   }

   read_first_on_file();

   stop_sign = 0;
```

```
    while((feof(drift_fp) == 0) && (stop_sign == 0))
    {
      read_coor();
      sort_vect_out_sum();
    }

    switch(frame_form)
    {
      case (1):
            get_mean_lp();
            break;
      case (2):
            get_mean_xy();
            break;
    }

    data.have_drift = true;
    data.total = res_vect;
    per_sample = res_vect / NR_OF_SAMPLES;
    data.per_day = per_sample * NR_OF_SAMP_A_DAY;

    fclose(drift_fp);
   }

   return data;
 }
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
/*utm.h*/
/*
* Module:         utm.h
* Purpose:        Converts degrees to UTM coordinates
* Author:Torstein Berge
*/
void gtu(double *x, double *y, double sm)
{
  static double ro  = 57.2957795;
  static double ra  = 6378388.00;
  static double e2  = 0.00672267;
  static double e2m = 0.00676817;
  static double f1a = 6365107.43;
  static double f1b = 16100.5912;
  static double f1c = 16.9694;
  static double f2a = 0.001174751;
  static double f3a = 0.00000023;
  static double f3b = 0.06091353;
  static double f3c = 0.000183233;
  static double f4a = 0.048461975;
  static double f5a = 0.000018985;

  static double fi,p,p2,sf,cf,cf2,tf,tf2,rn,cn,scn,f1,f2,f3,f4,f5;
  static int ns;

  ns = 0;
  if( *y < 0.0 ) {
    ns = -1;
    *y = -(*y);
  }
  fi  = (*y)/ro;
  p   = ( (*x) - (sm) )*0.36;
  p2  = p*p;
  sf  = sin( fi );
  cf  = cos( fi );
  cf2 = cf * cf;
  tf  = sf/cf;
  tf2 = tf*tf;
  rn  = ra/sqrt(1.0 - e2*sf*sf);
  cn  = rn*cf;
  scn = cn*sf;
  f1  = f1a*fi - f1b*sin( fi*2.0 ) + f1c*sin( fi*4.0 );
  f2  = f2a*scn;
  f3  = f3a*scn*cf2*( 5.0-tf2+cf2*(f3b+f3c*cf2));
  f4  = f4a*cn;
  f5  = f5a*cn*cf2*( 1.0-tf2+e2m*cf2);

  *y  = f1+p2*(f2+f3*p2);
  *x  = 500000.0 + p*(f4+f5*p2);
  if(ns < 0 ) *y = 10000000.0 - *y;
}


void utg(double *x, double *y, double *sm, int *ns)
{
  static double ro  = 57.2957795;
  static double ra  = 6378388.0;
  static double e2  = 0.00672267;
  static double fb0 = 0.000000156;
  static double fsa = 6365107.4;
  static double fsb = 16100.59;
  static double fsc = 16.97;
  static double f6a = 2.8670822e13;
  static double f6b = 1.9404900e11;
  static double f7a = 1.1955738e25;
  static double f7b = 7.1734431e24;
  static double f7c = 9.7102165e22;
  static double f7d = 3.2860198e20;
  static double f7e = 9.8580594e20;
  static double f8a = 5.7318707e7;
  static double f9a = 9.5607649e18;
  static double f9b = 1.9121530e19;
  static double f9c = 6.4708882e16;
  static double fea = 2.3921045e30;
  static double feb = 1.3395785e31;
  static double fec = 1.1482102e31;
  static double fed = 1.9428204e28;
  static double fee = 2.5904272e28;
  static double b0,s,sxn,q,qq,sf,sf2,cf,cf2,tf,tf2,rn,rn2,tfrn2,rncf;
  static double f6,f7,f8,f9,fe;

  if( *ns < 0 ) *y = 10000000.0 - (*y);
  b0  = fb0 * (*y);
  do {
    s   = fsa*b0 - fsb*sin( b0*2.0 ) + fsc*sin( b0*4.0 );
    sxn = s - *y;
    b0  = b0 - fb0*sxn;
  } while ( fabs(sxn) > 0.1 );

  q    = (*x)/1000000.0 - 0.5;
  qq   = q*q;
  sf   = sin( b0 );
  sf2  = sf*sf;
  cf   = cos( b0 );
  cf2   = cf*cf;
  tf    = sf/cf;
  tf2   = tf*tf;
  rn    = sqrt( 1.0 - e2*sf2)/ra;
  rn2   = rn*rn;
  tfrn2 = tf*rn2;
  rncf  = rn/cf;

  f6 = (f6a+f6b*cf2)*tfrn2;
  f7 = (f7a+f7b*tf2+f7c*(cf2-sf2)-f7d*cf2*cf2-f7e*cf2*sf2)*tfrn2*rn2;
  f8 = f8a*rncf;
  f9 = (f9a+f9b*tf2+f9c*cf2)*rncf*rn2;
  fe = (fea+feb*tf2+fec*tf2*tf2+fed*cf2+fee*sf2)*rncf*rn2*rn2;

  (*y) = ro*b0 + qq*(-f6 + qq*f7);
  (*x) = *sm    +  q*( f8 + qq*(-f9 + qq*fe));
  if( *ns < 0 ) (*y) = -(*y);
}
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

Data formats

Data from the ULS is stored continously in sequential Seadata format. Counts from the transducers, for echo one and two and water pressure, are stored, followed by the tilt in degrees. The data is separated in blocks of constant size. This can lead to occational blocks that are not completelly filled.

The original data file is konverted to ASCII-format in Bergen and to tabulated form by a Fortran programme implemented at NP. The original data file is split into separate files, each corresponding to one calendar month. There is one sample per line. Normally this represents continous four-minute samples (in the used material from 1988). Each line contains time, counts for the two most equal echo times, count from the depth transducer and tilt. Directly after the tilt value is sometimes a star (asterisk), marking values with lower or unknown precission. There can also be text comments after the data items and possilble star marking. Time is in the form yy:ddd:mmmm. Days are in Julian day number (1-365, 366 for leap years) and minutes are in minute number in the day (1 -1440). For example 4:12 pm 9:th of June 1988 will be 88:161:0972. (1988 is a leap year.)

The air pressure data file is in a similar format with comments after the data values on some lines. The air pressure used is read from weather maps manually and keyed into a file in ASCII format. The pressure at 00Z and 12Z is used and stored after year and Julian day number. The unit is mbar = hPa. For the period corresponding to the ULS measurements not all days have air pressure values as some maps were missing. This is marked with comments in the file.

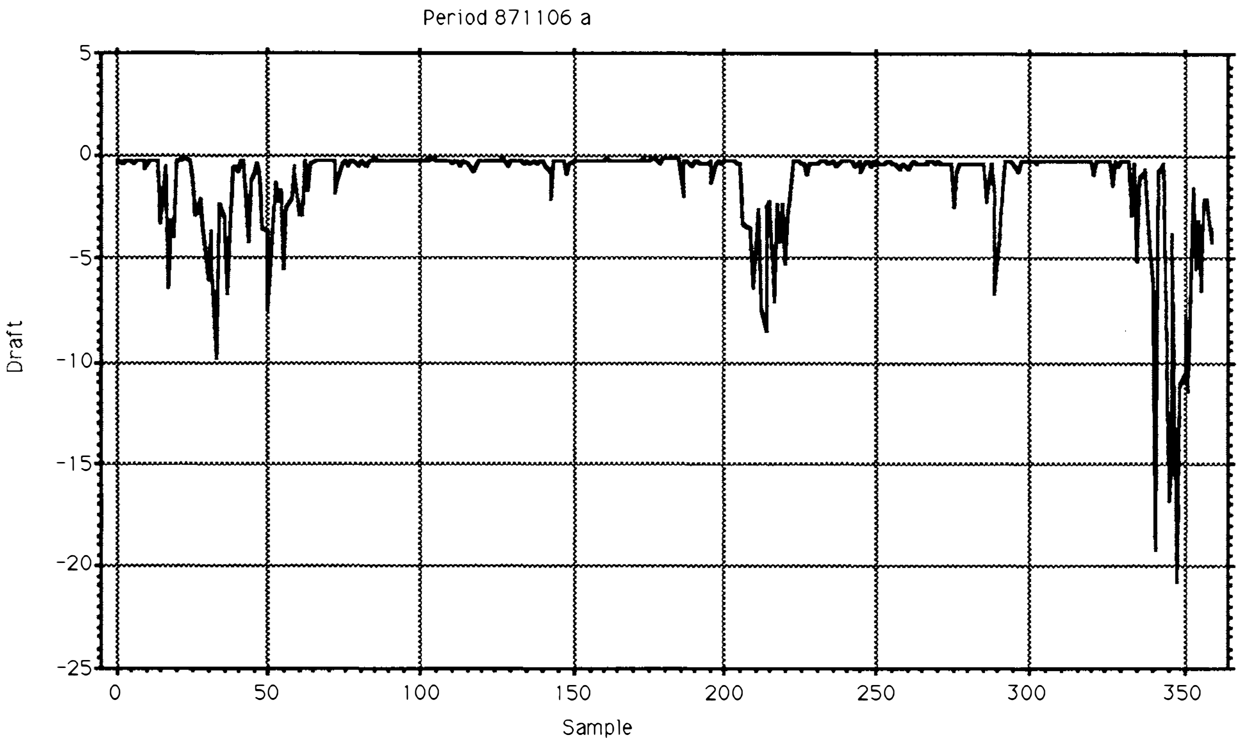
Temperature data is stored in separate files for different depths. The measurements close to the ULS 1987-88 was unfortunatelly not useable. Temperature data from 72 resp. 101 meters below sea surface was kindly supplied by Alfred Wegener Institute for Polar and Marine Research in Bremerhafen, Germany. The data is daily means at about these depths. In the 72 m. b. s. s. file are also current values included. Currents are represented as N-S- and E-W-component, as well as as speed and direction.
First in the file is a block of descriptive data, for example giving depth at mooring site, position, instrument and header text.

The vector field coordinate files can at present have three different formats, ARC/INFO, "Zhang"-format or NP-format. First in the files are one or more lines that should be skipped. In the ARC/INFO files each vector is represented with an ID-number followed by the start-coordinates and the end-coordinates on consequtive lines, and with an END last.
In the "Zhang" files each of the data lines contains a type identifier first on the line, a P normally and an E for the last record. After the E usually there'll be an A first on the next line. It's a block of statistics starting with Average. After the type identifier lies latitude and longitude for the start- and endcoordinates, on the same line, followed by u- and v-components, displacements and direction. NP-format is a digital map format used at the Norwegian Polar Research Institute.

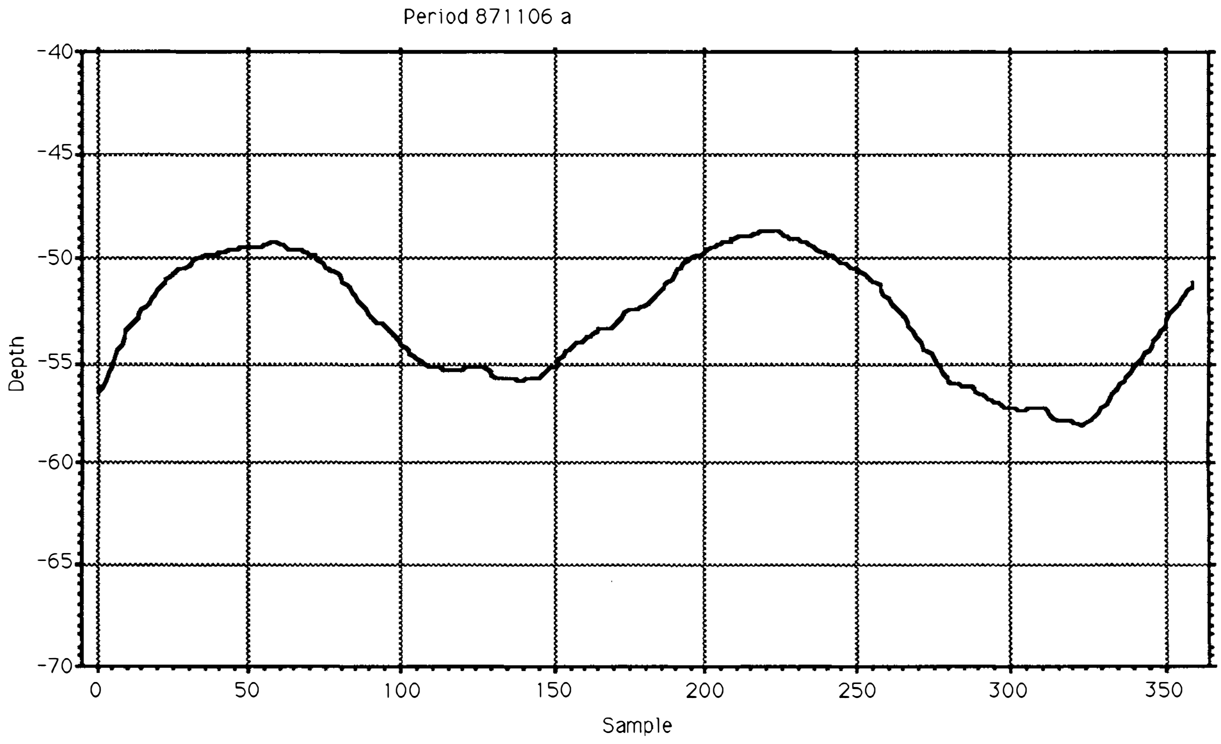
The coordinates should be in decimal latitude and longitude. Later maybe also other units will be accepted. The computations are made in global coordinates and in UTM.

The AVHRR images from TSS is sent on computer compatible tape (CCT) of a density of 6250 bpi. Normally a tape contains four images, but by technical reasons it may be fewer.
The scenes are stored in BIL-format (Bit Interleaved by Line). That means that the line with the same line number in each channel lays on a train before the next line of each channel in a similar way etc. Before channel one there is a leader and after channel five there is a trailer.
The data is stored in 16 bits per pixel and pixel 4, 5 and 6 are used for global grid net, land contours and national boundaries. On NOAA-9 images channel 5 is missing and channel 3 is distorted in 1987-88.

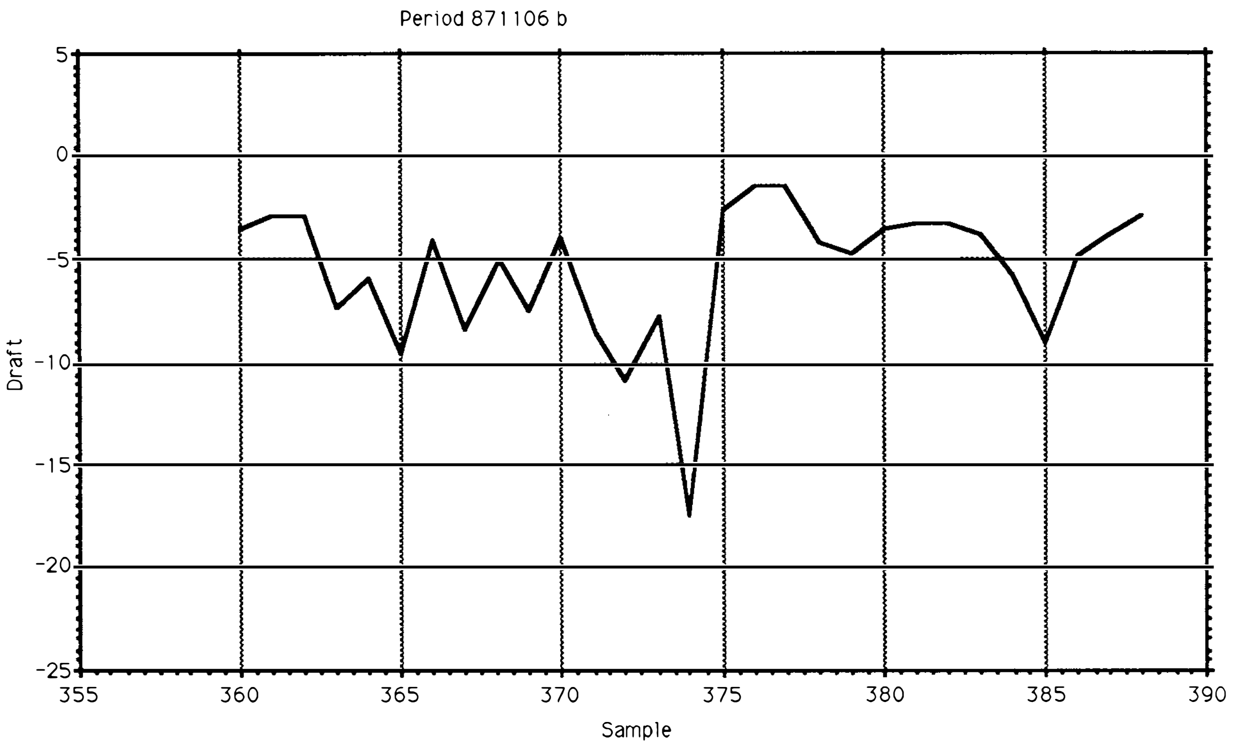
Draft- and uls depth diagrams for the periods with AVHRR images.
Periods about 24 h or longer are shown in 24 h units.



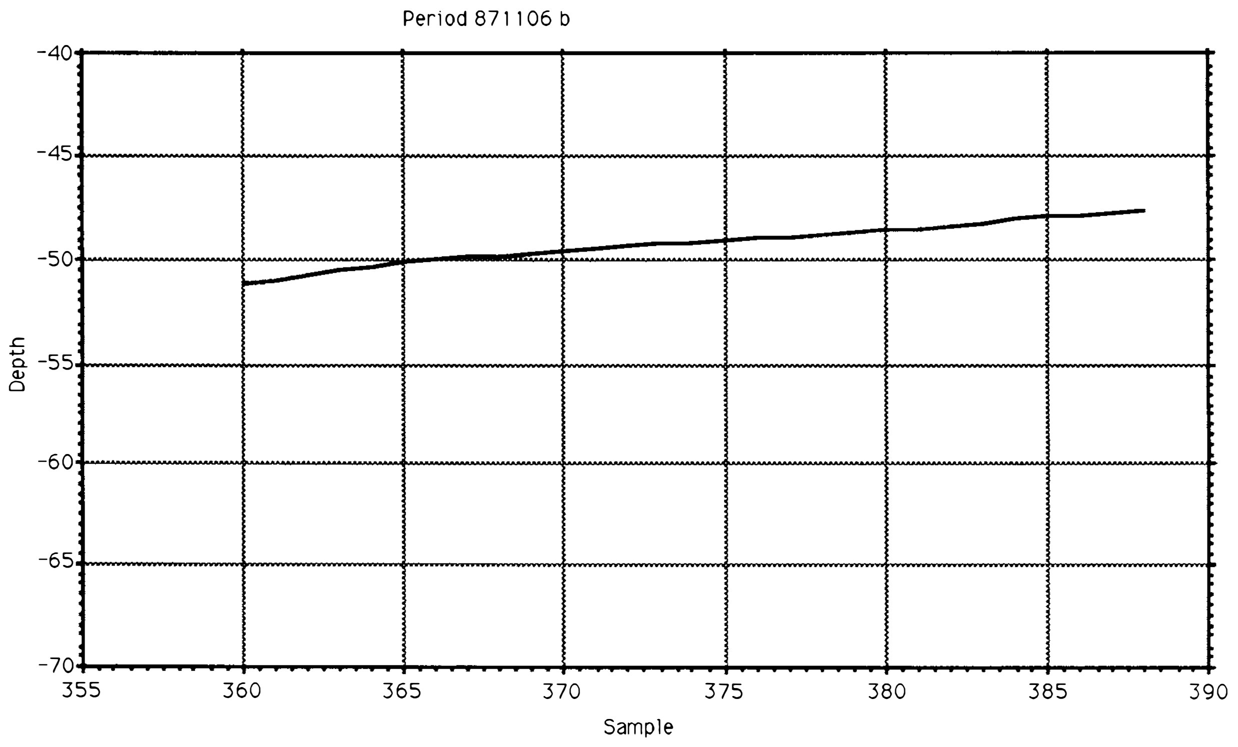
Period 871106 a, draft.



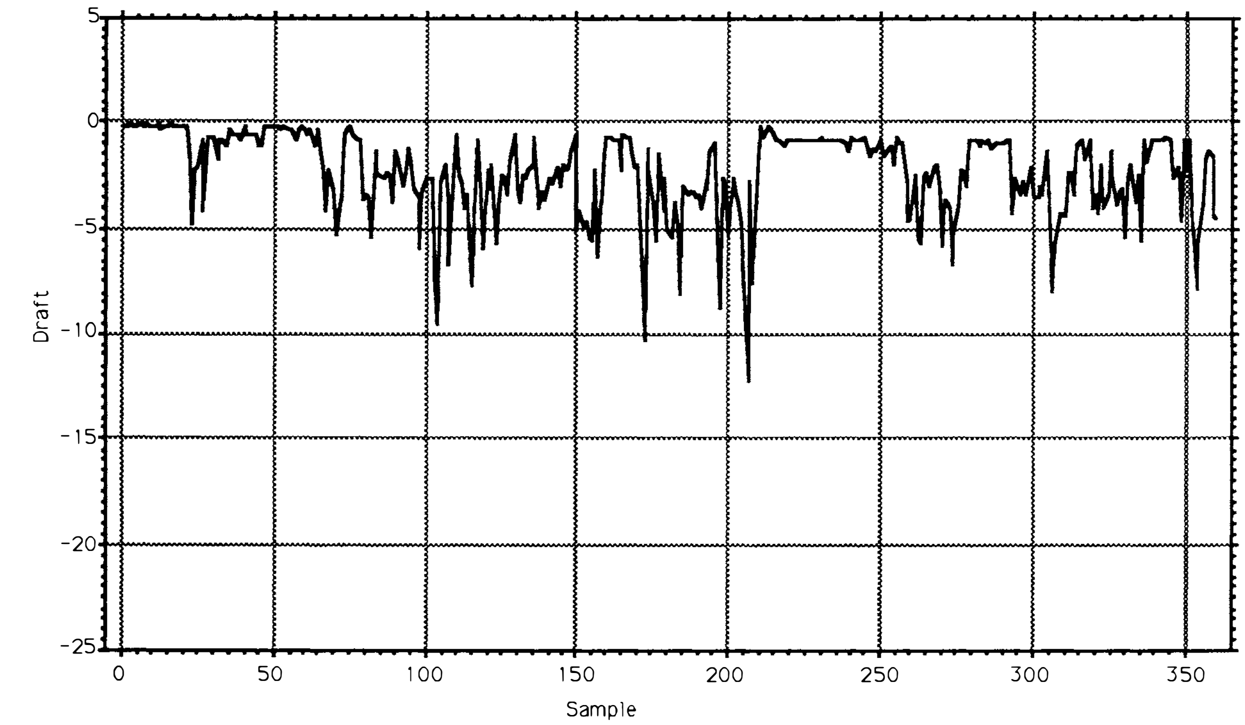
Period 871106 a, uls depth.



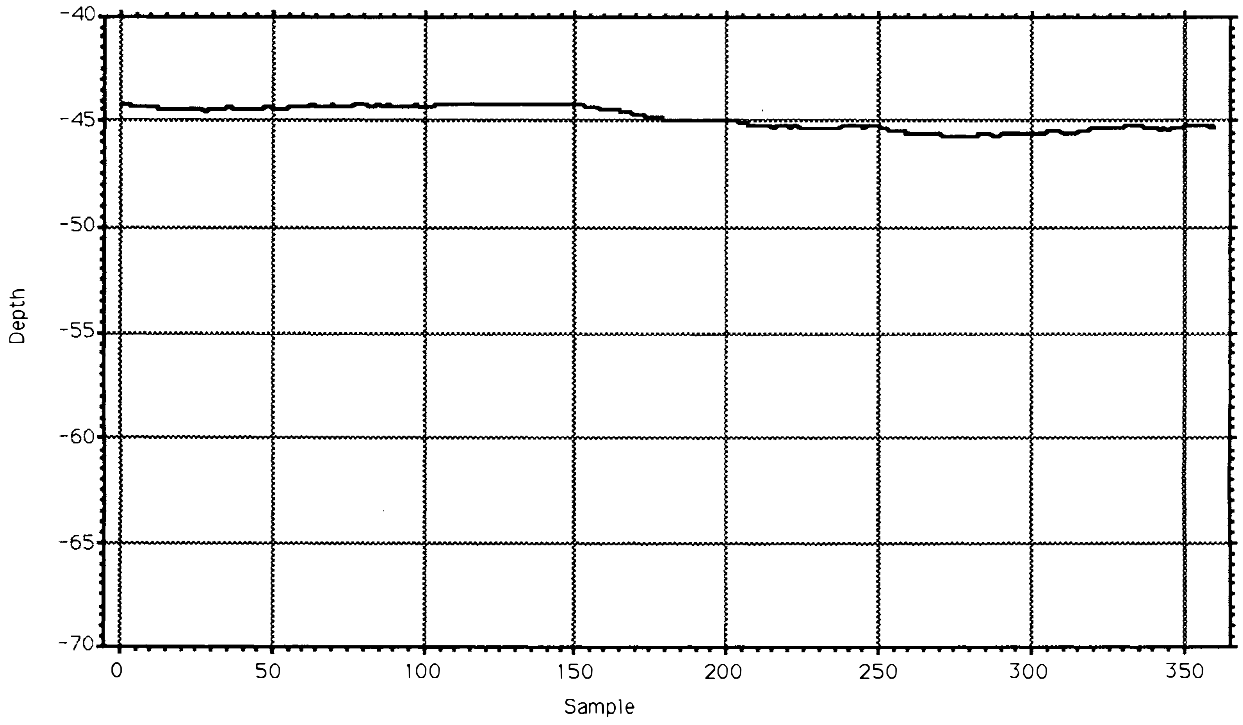
Period 871106 b, draft.



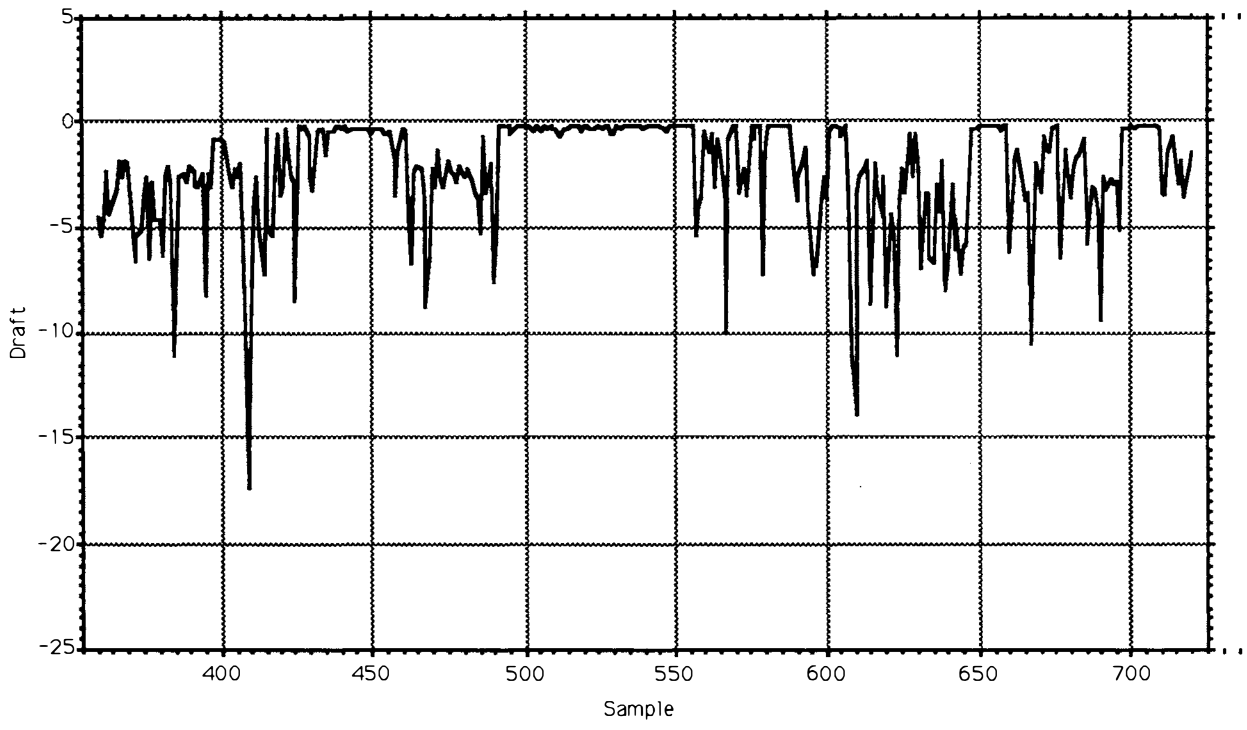
Period 871106 b, uls depth.



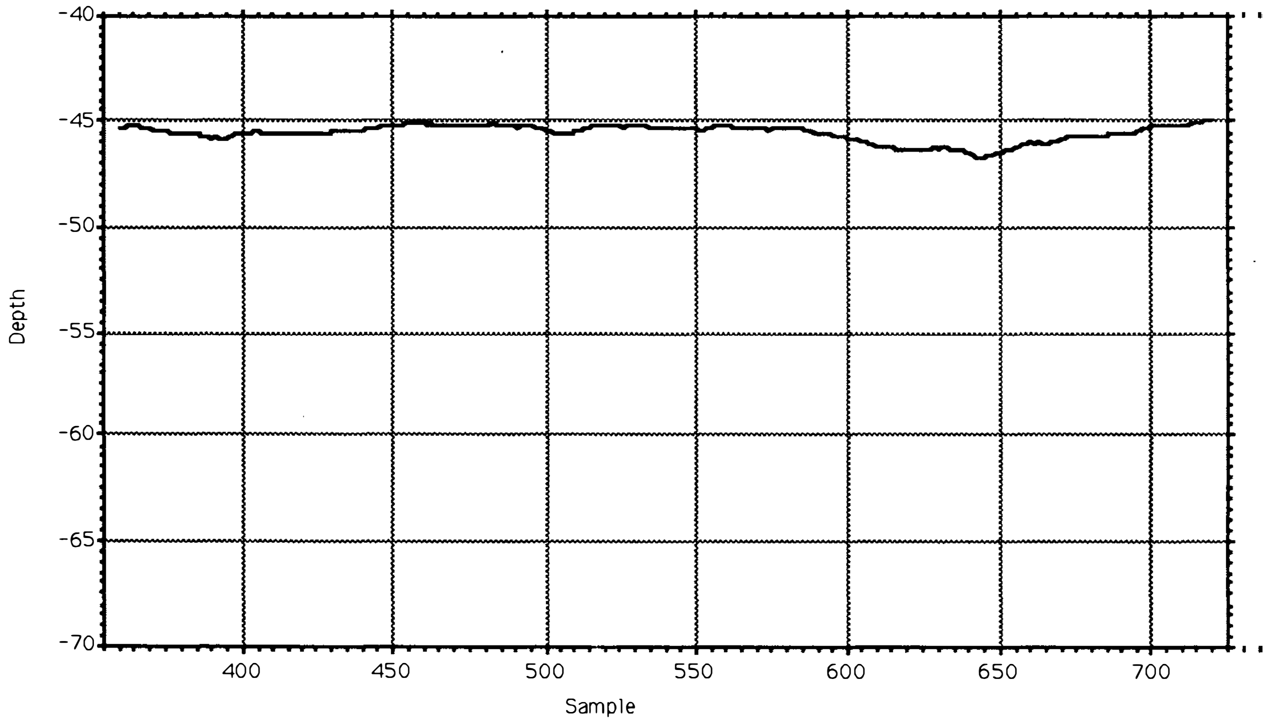
Period 871116 a, draft.



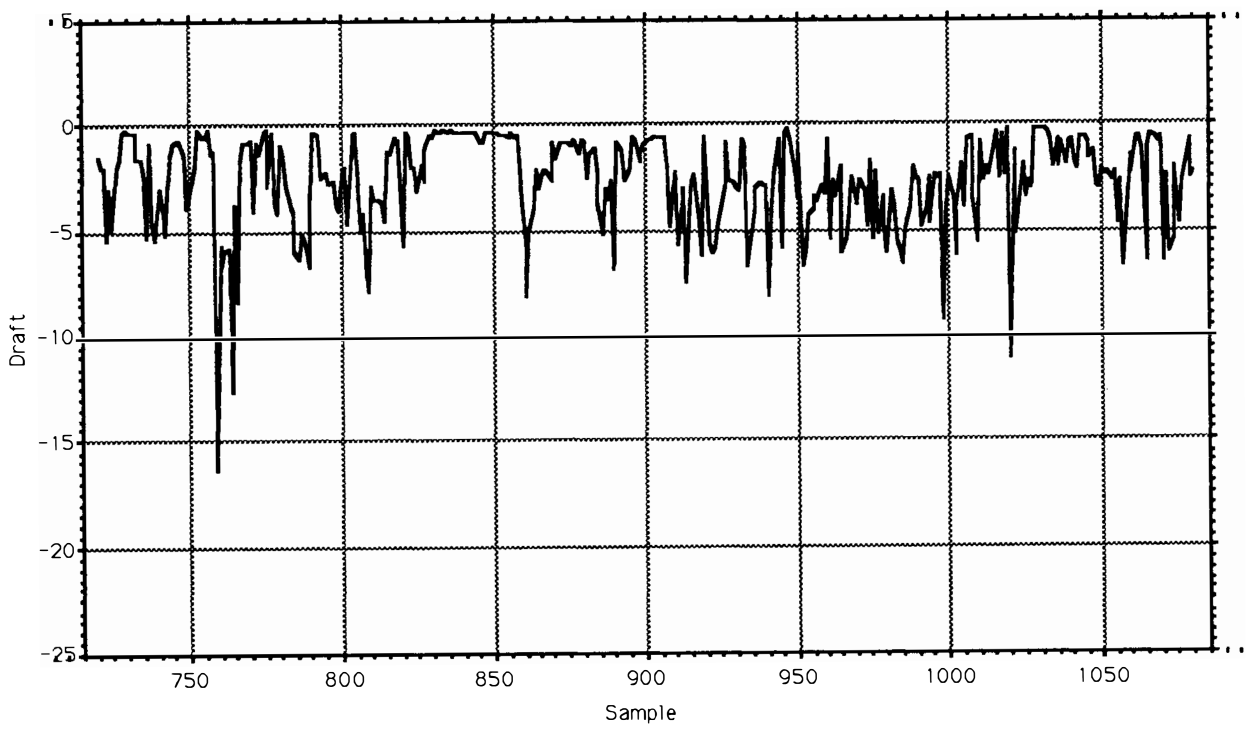
Period 871116 a, uls depth.



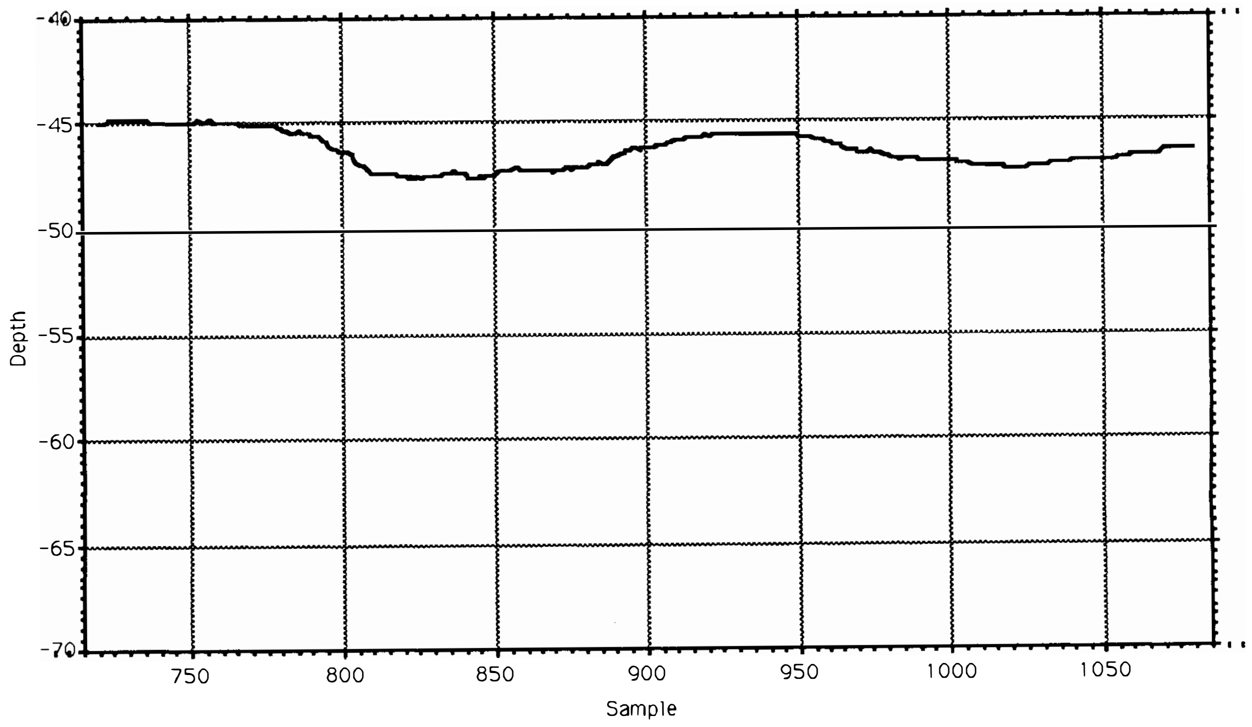
Period 871116 b, draft.



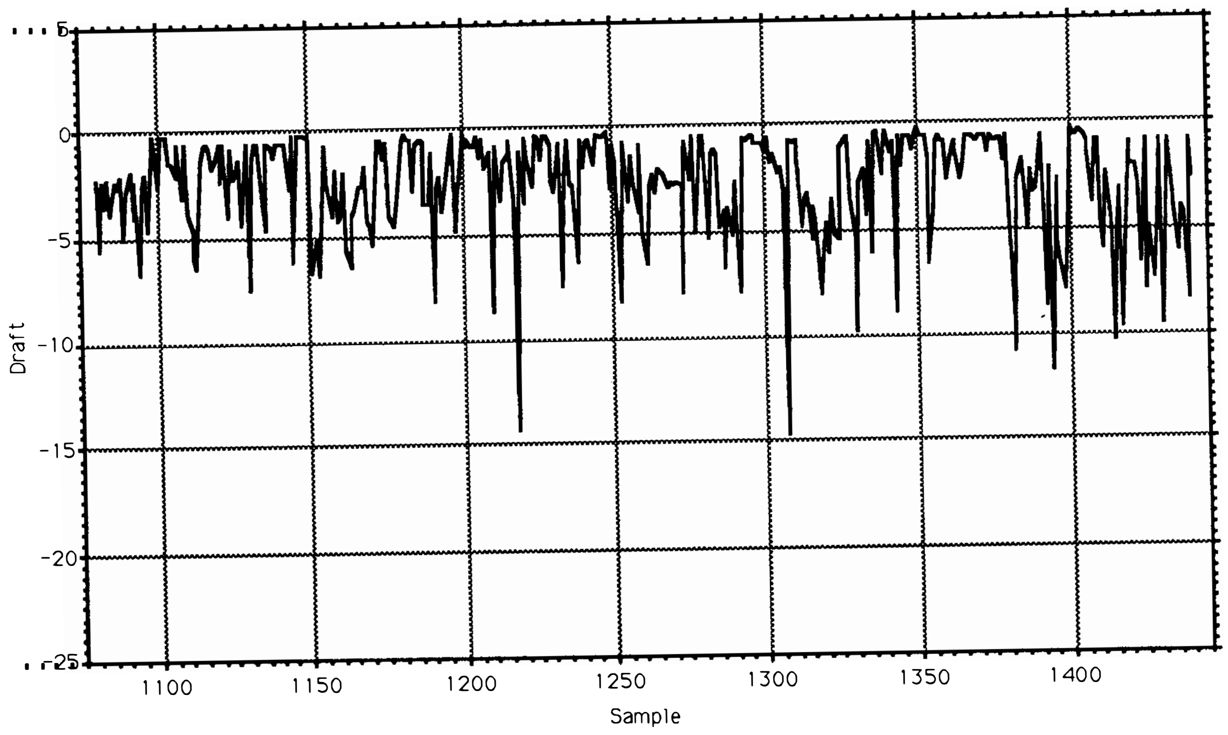
Period 871116 b, uls depth.



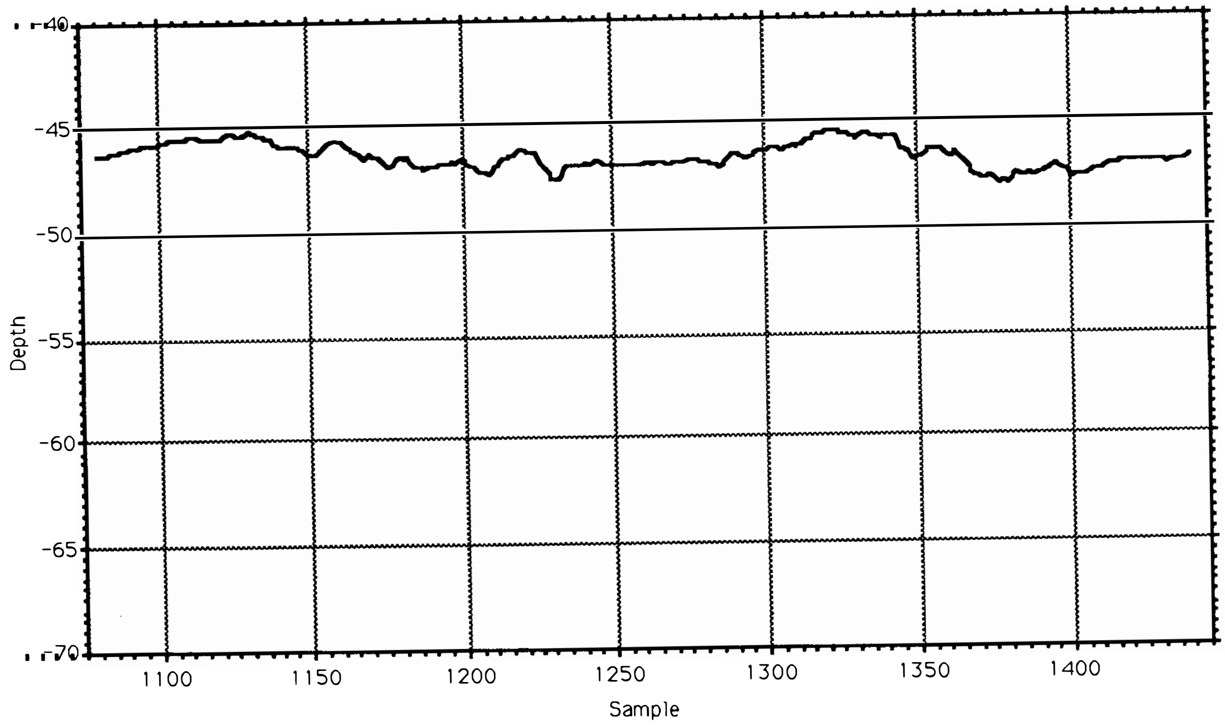
Period 871116 c, draft.



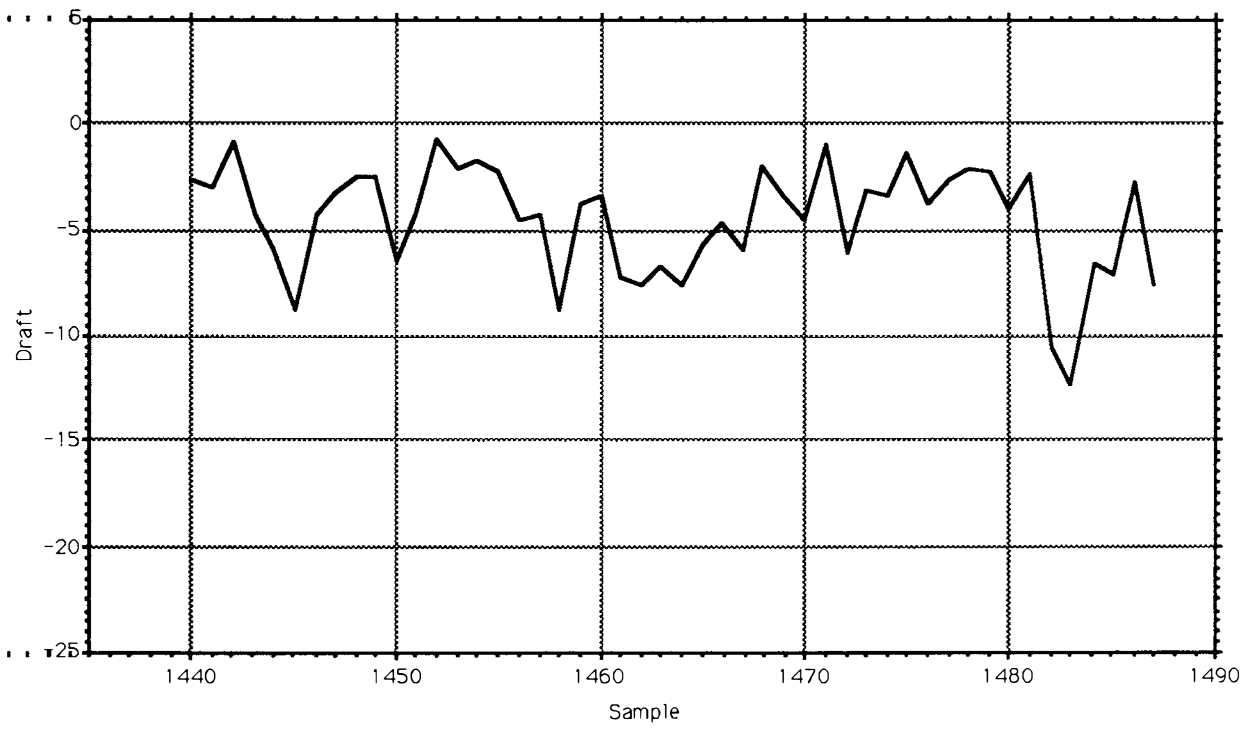
Period 871116 c, uls depth.



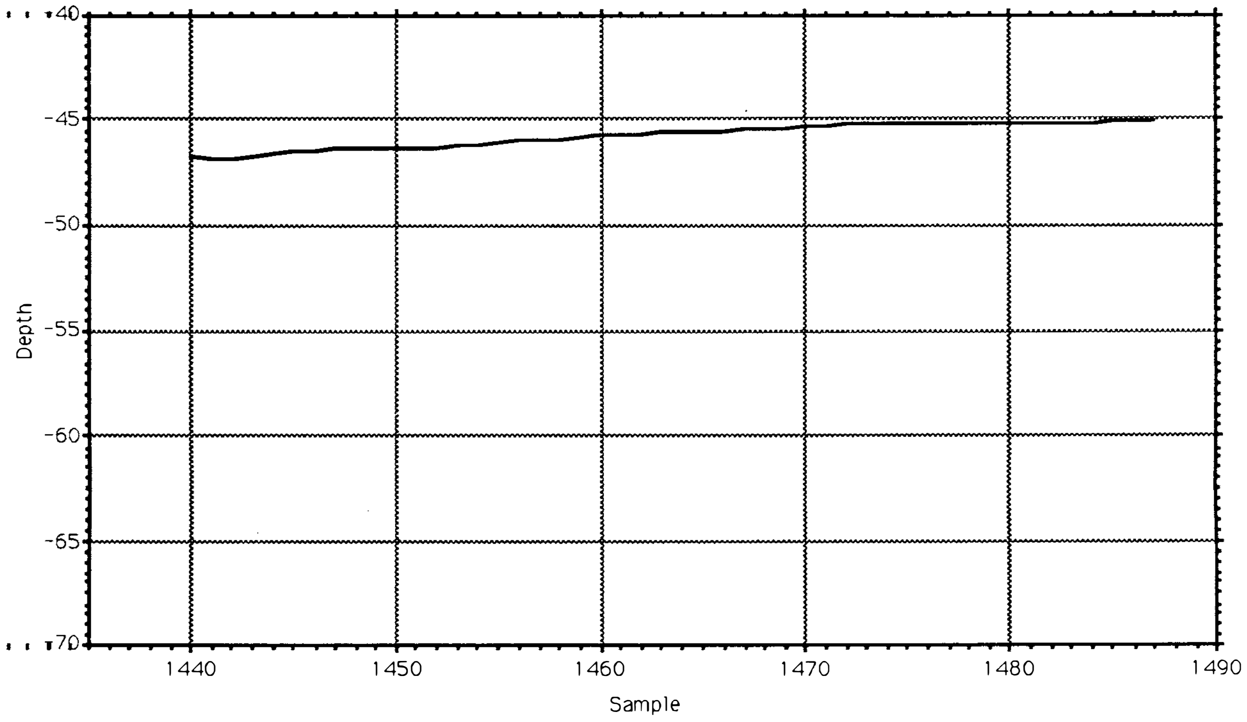
Period 871116 d, draft.



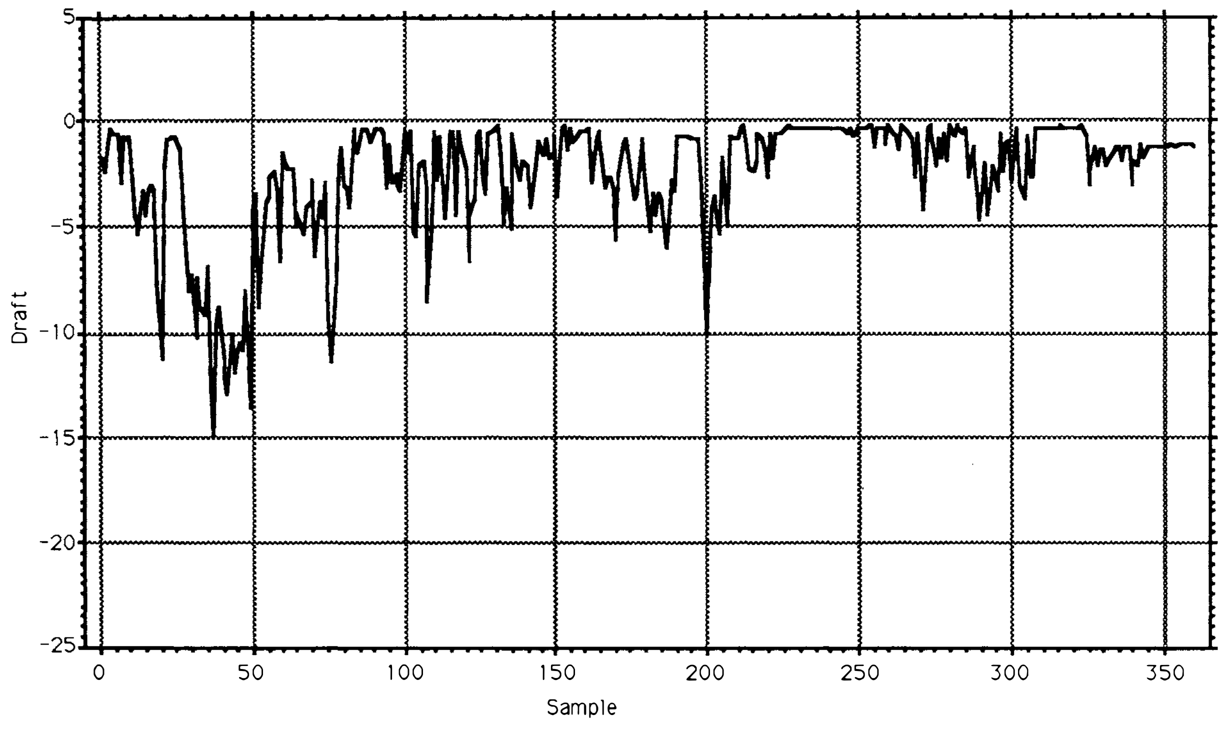
Period 871116 d, uls depth.



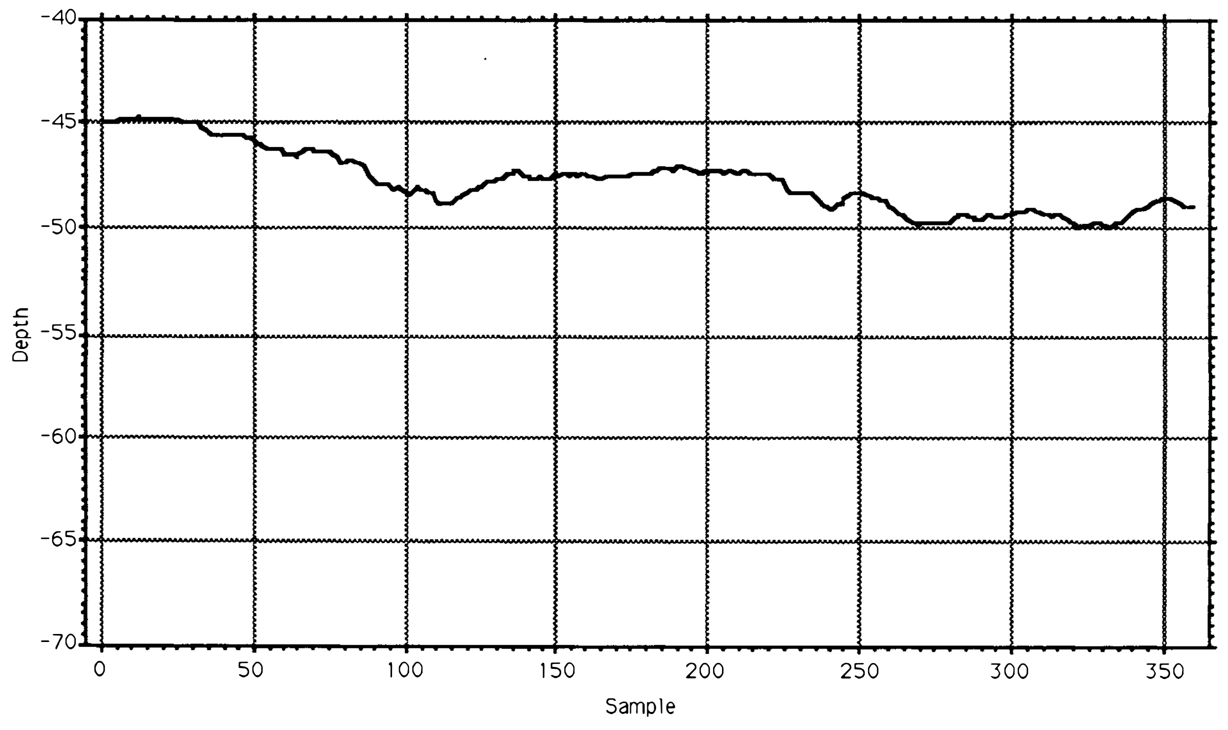
Period 871116 e, draft.



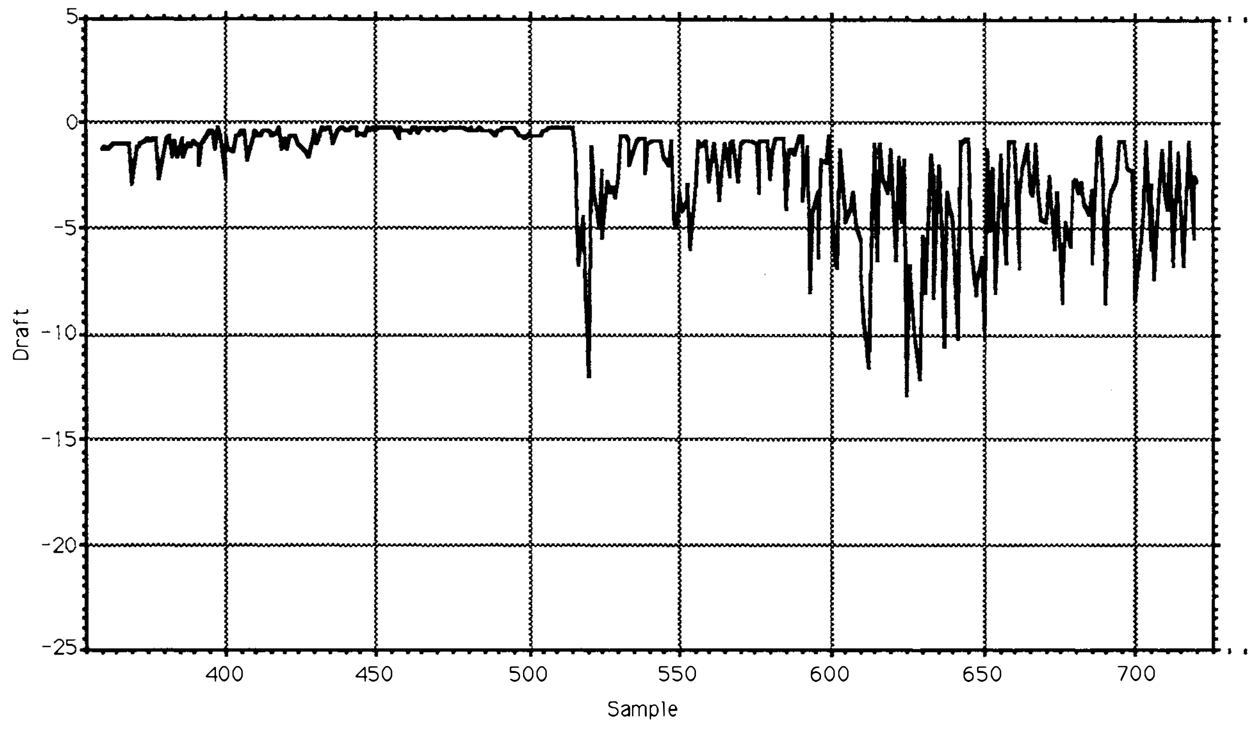
Period 871116 e, uls depth.



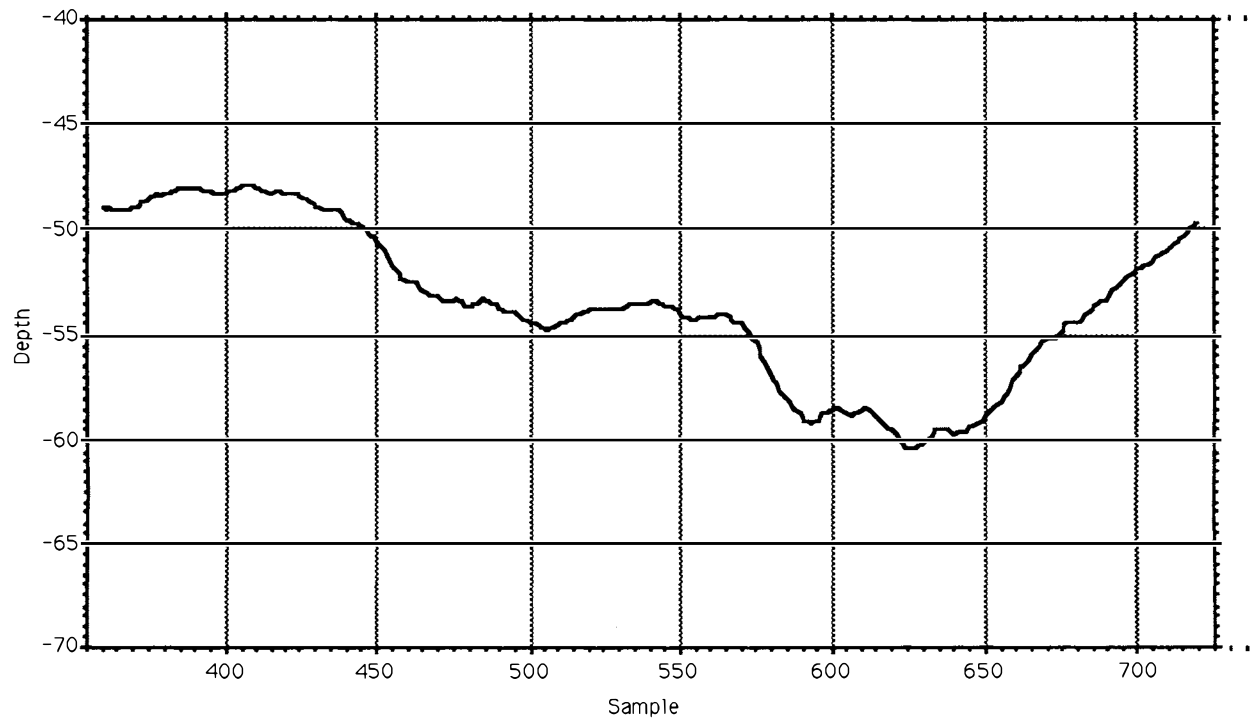
Period 871120 a, draft.



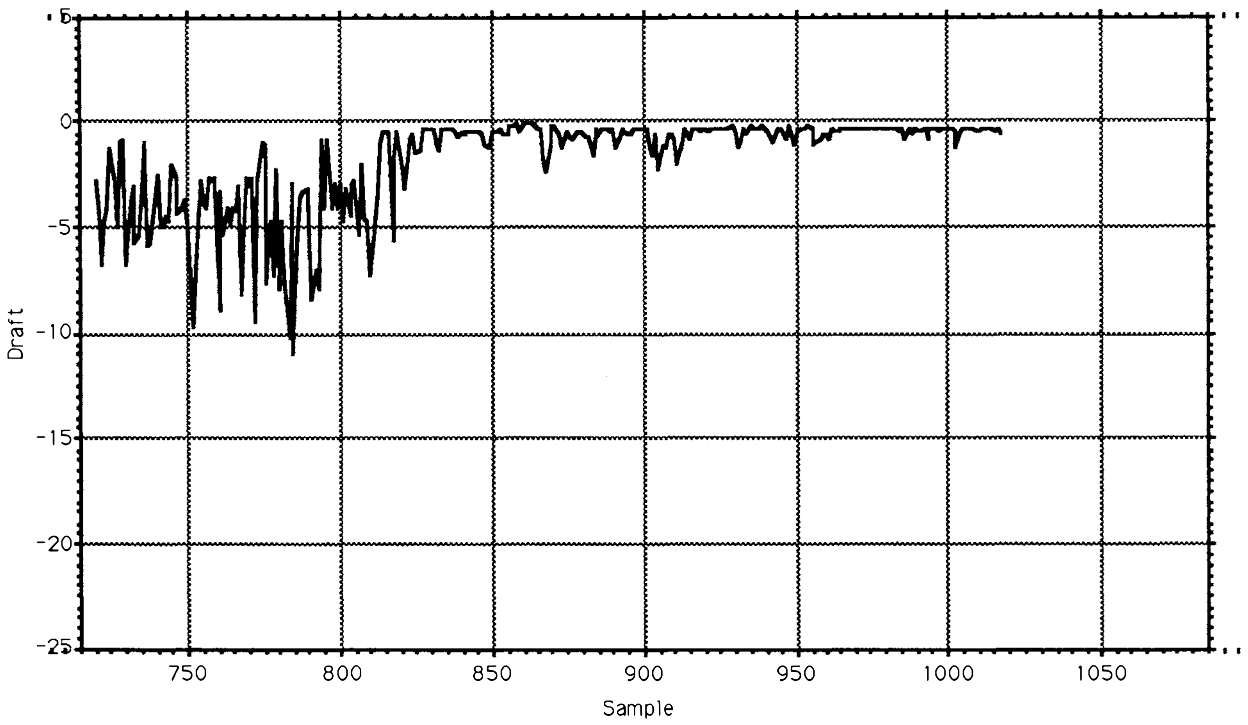
Period 871120 a, uls depth.



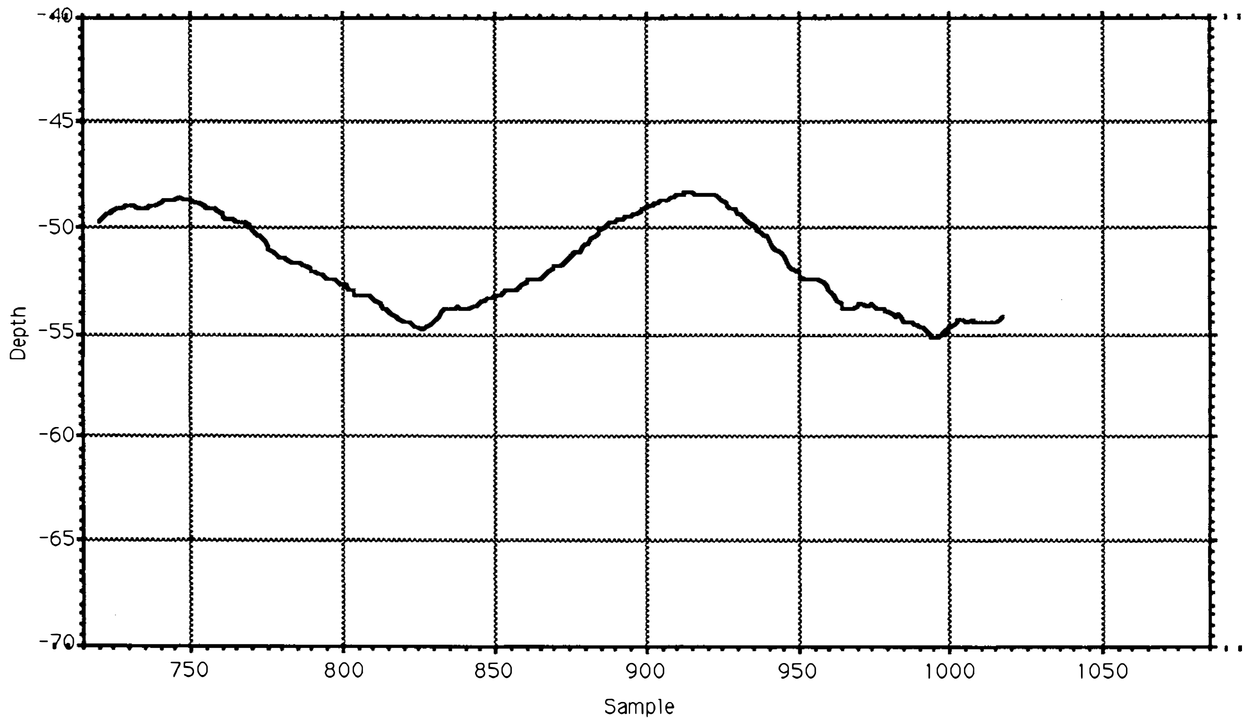
Period 871120 b, draft.



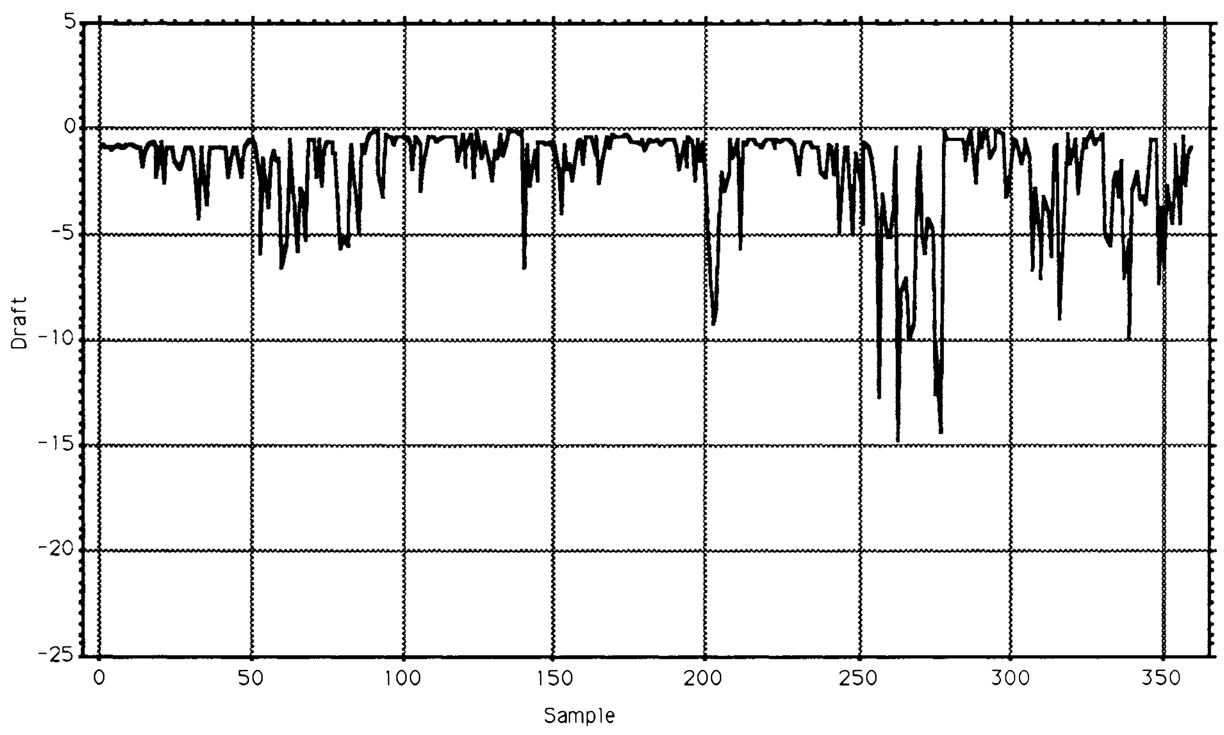
Period 871120 b, uls depth.



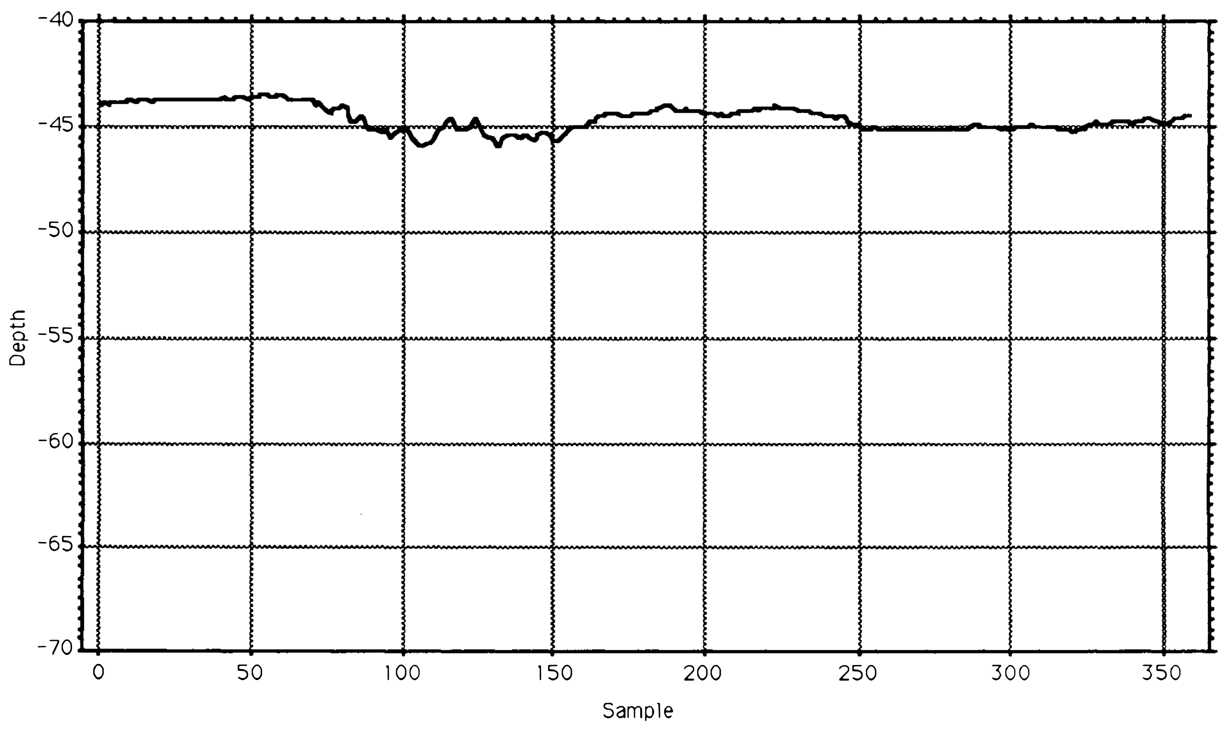
Period 871120 c, draft.



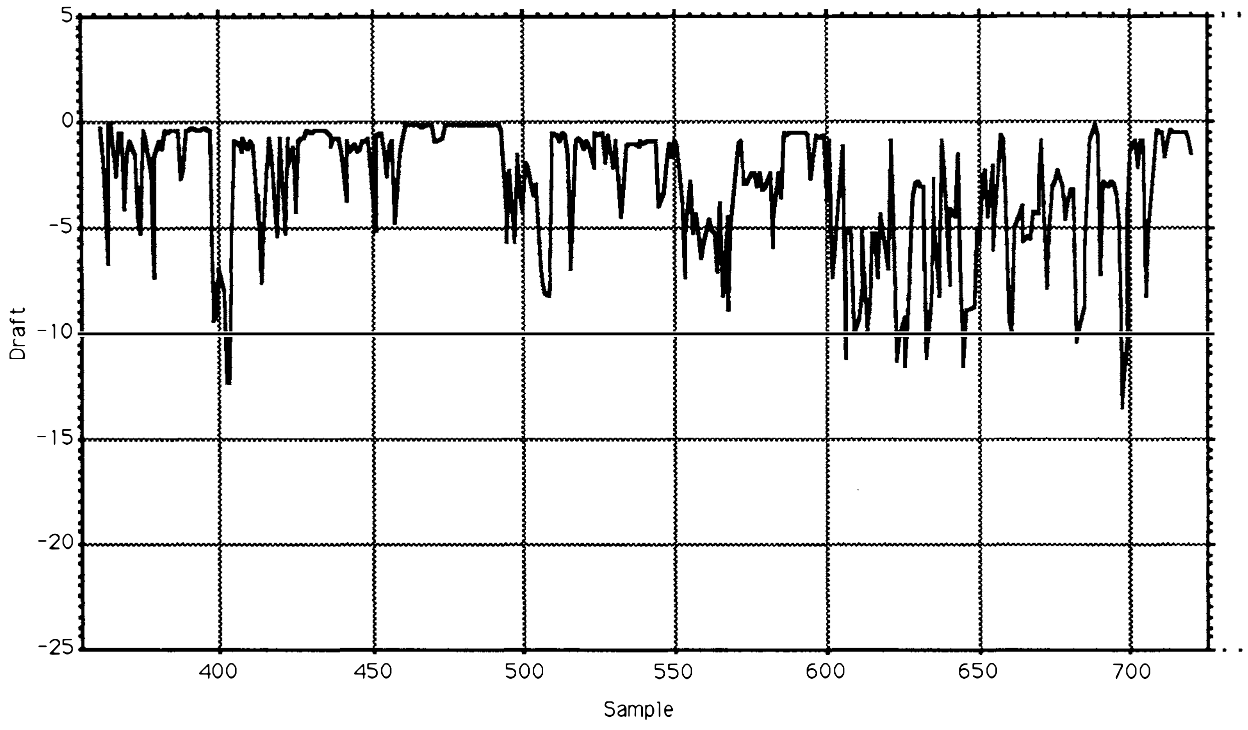
Period 871120 c, uls depth.



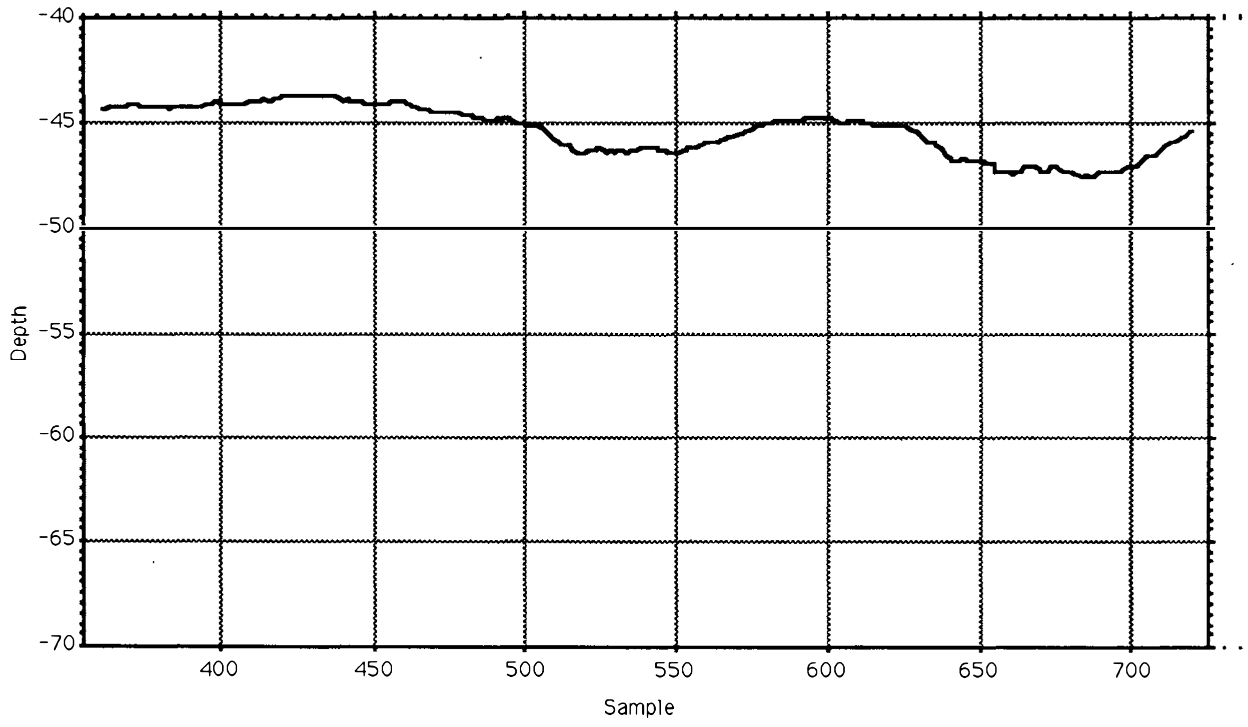
Period 871204 a, draft.



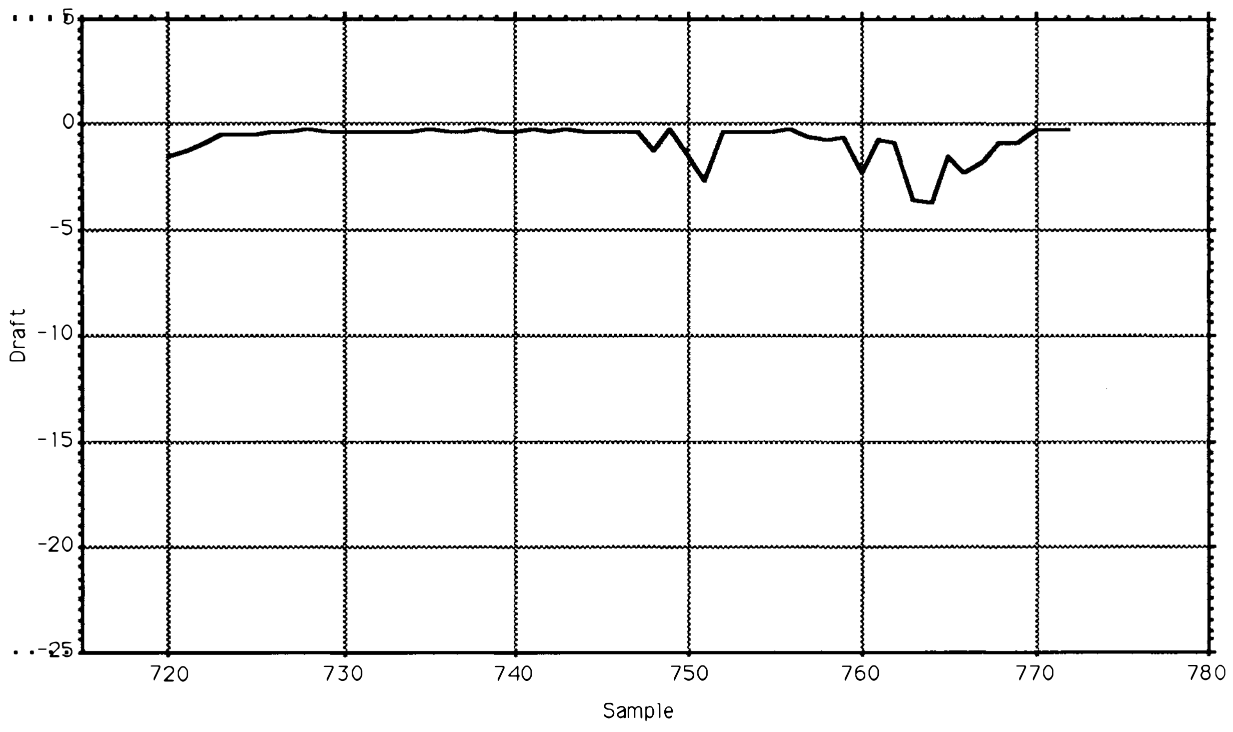
Period 871204 a, uls depth.



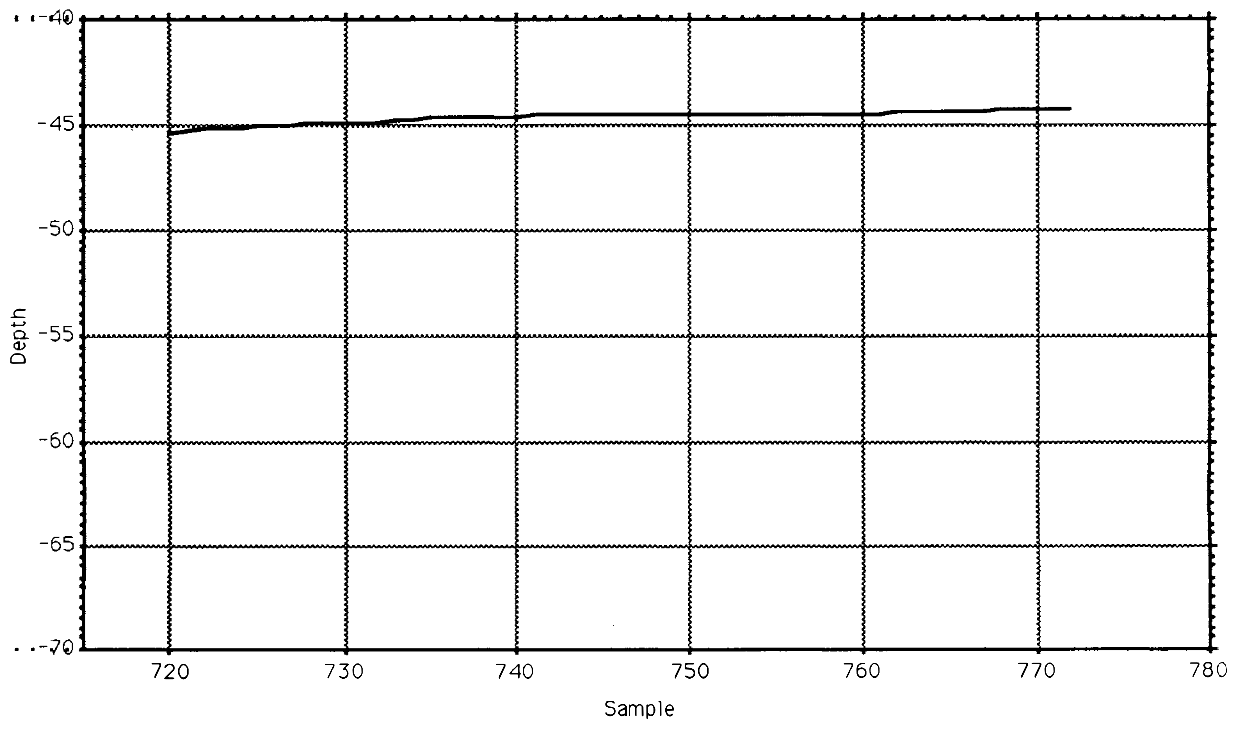
Period 871204 b, draft.



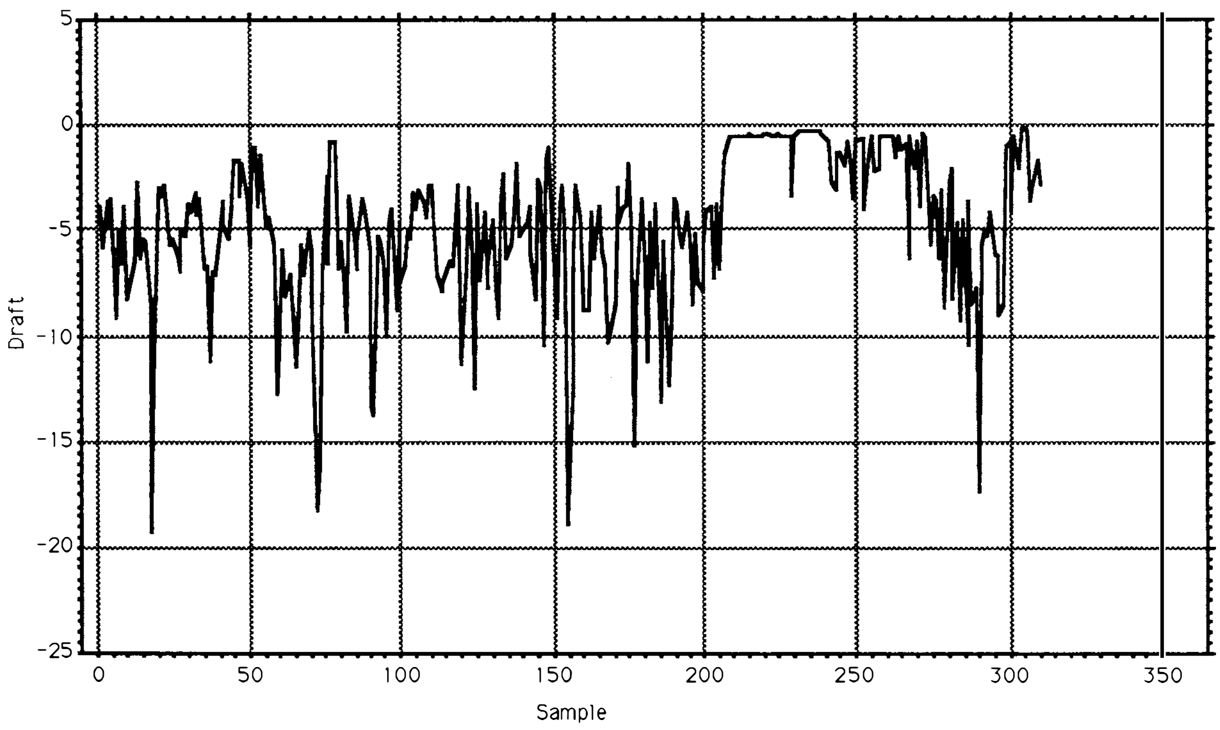
Period 871204 b, uls depth.



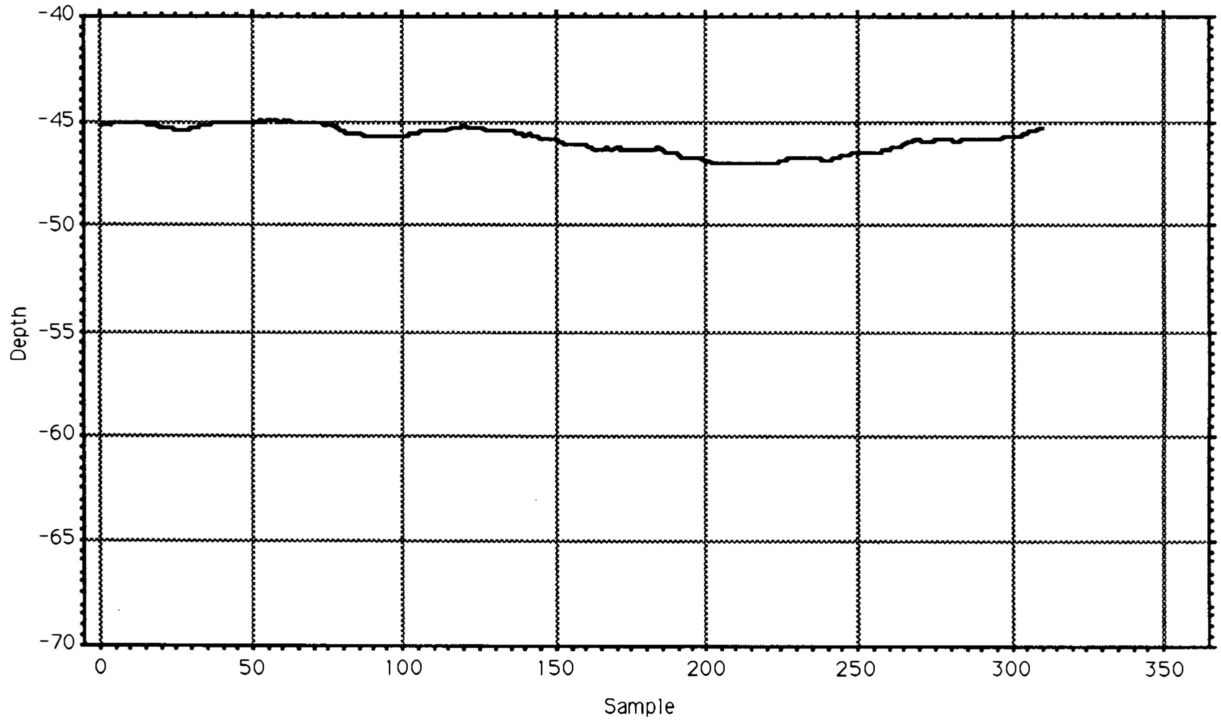
Period 871204 c, draft.



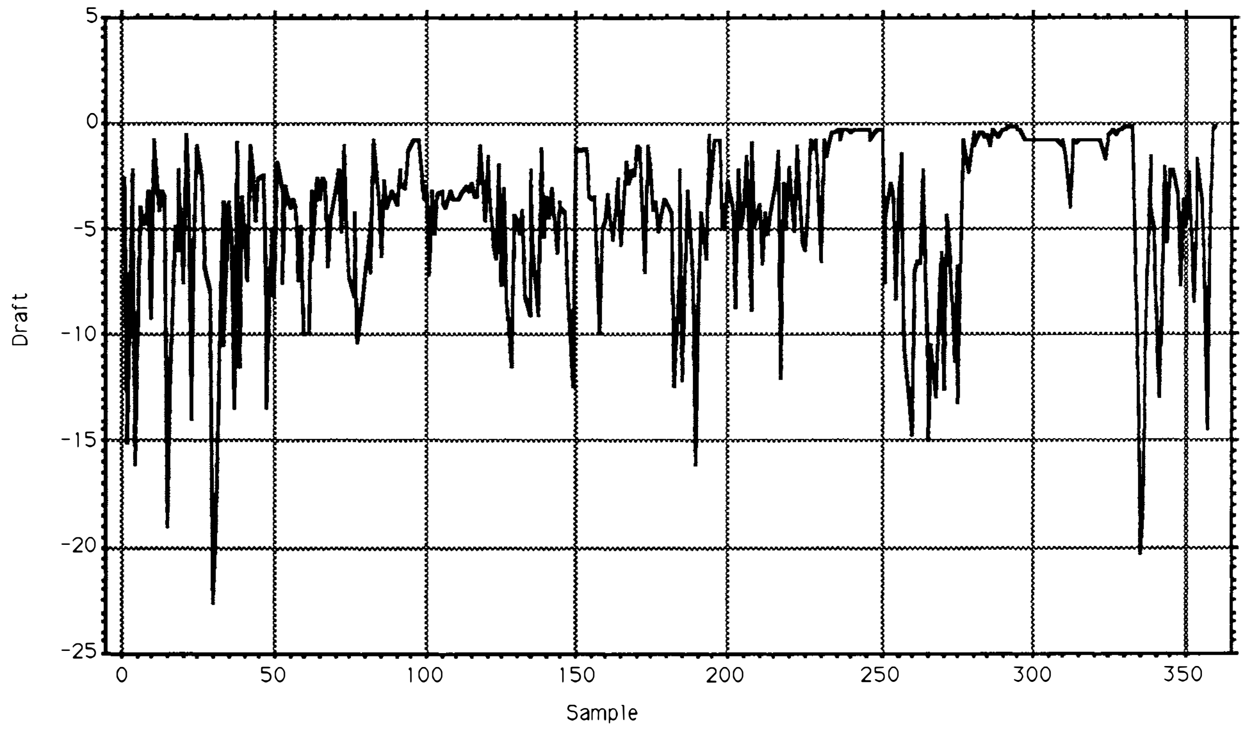
Period 871204 c, uls depth.



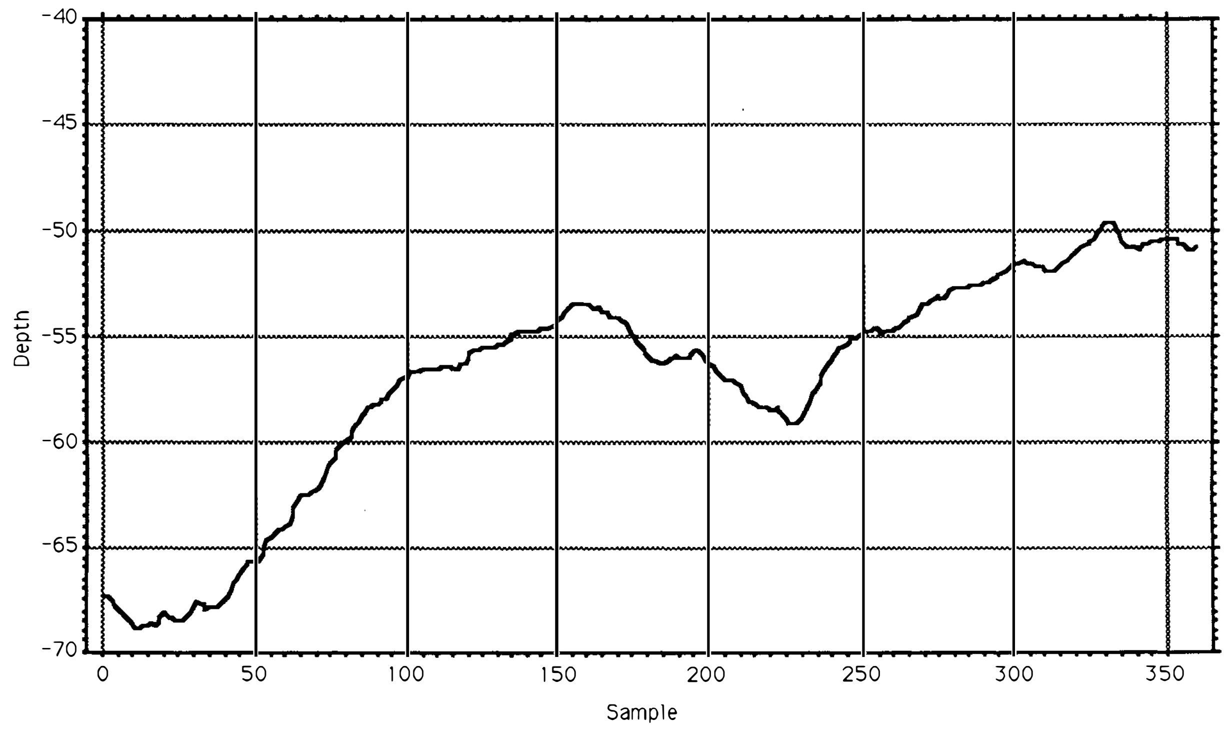
Period 871215 a, draft.



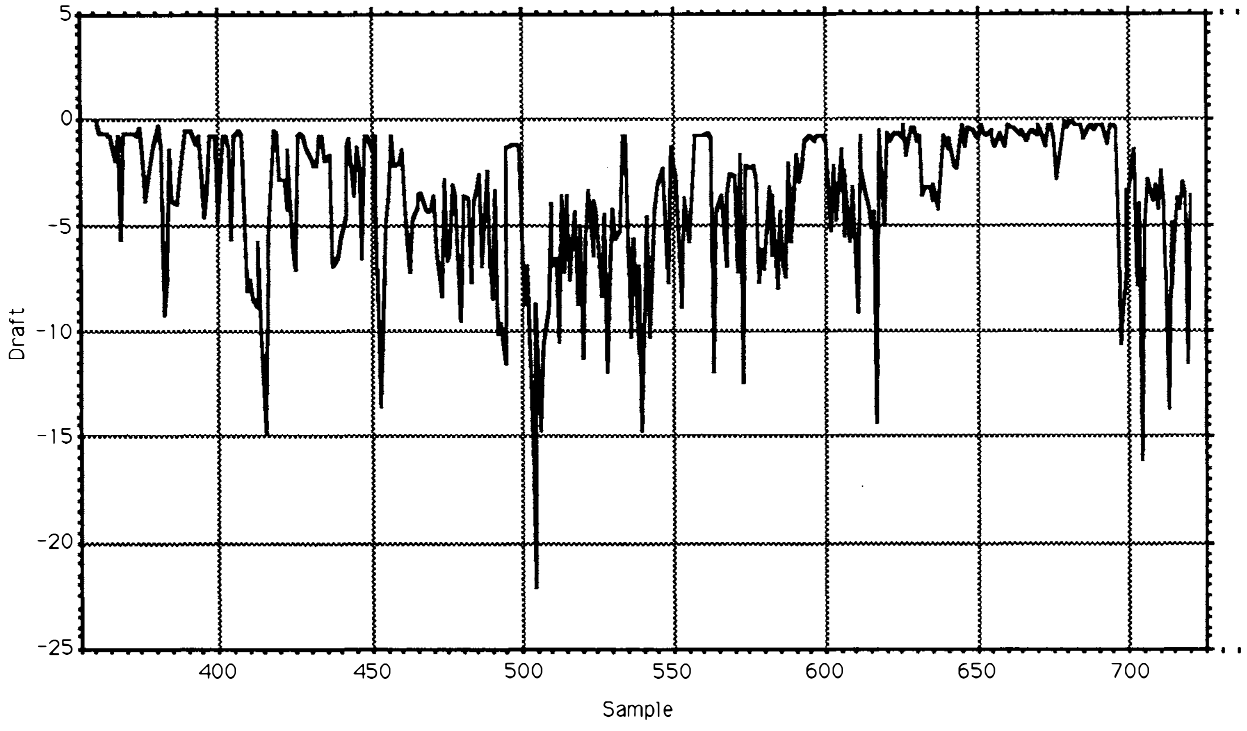
Period 871215 a, uls depth.



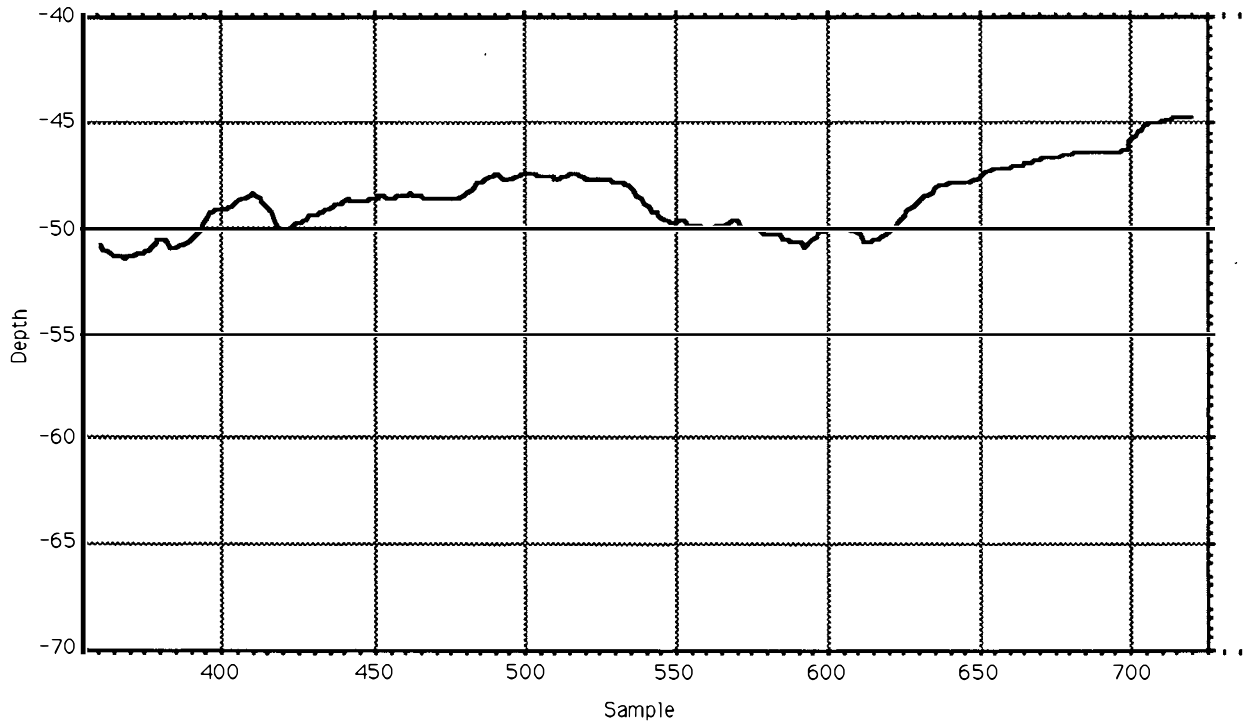
Period 871227 a, draft.



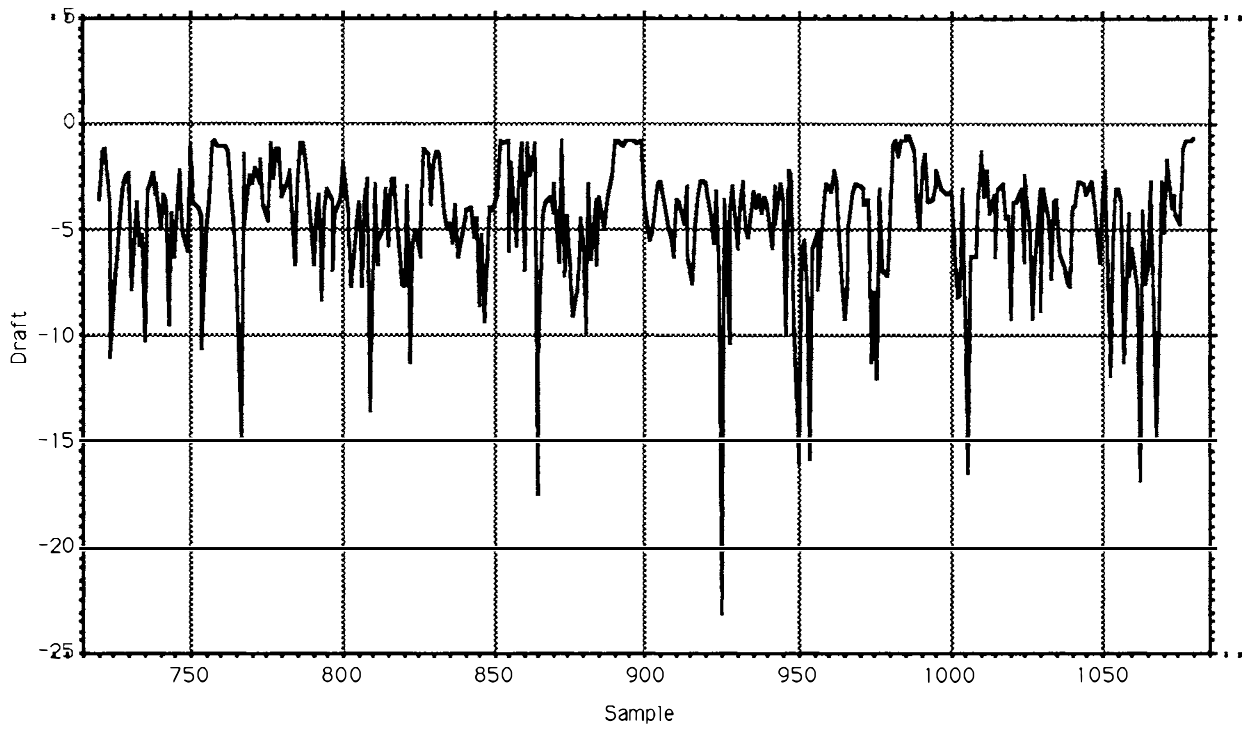
Period 871227 a, uls depth.



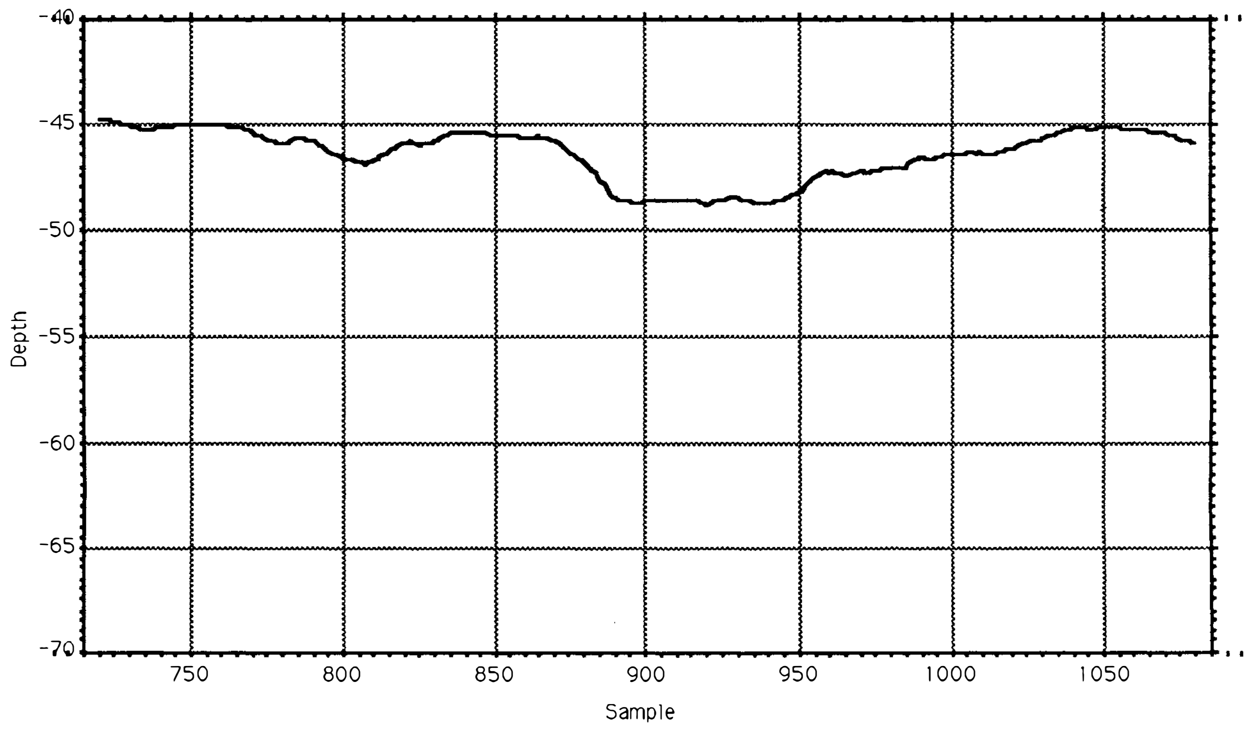
Period 871227 b, draft.



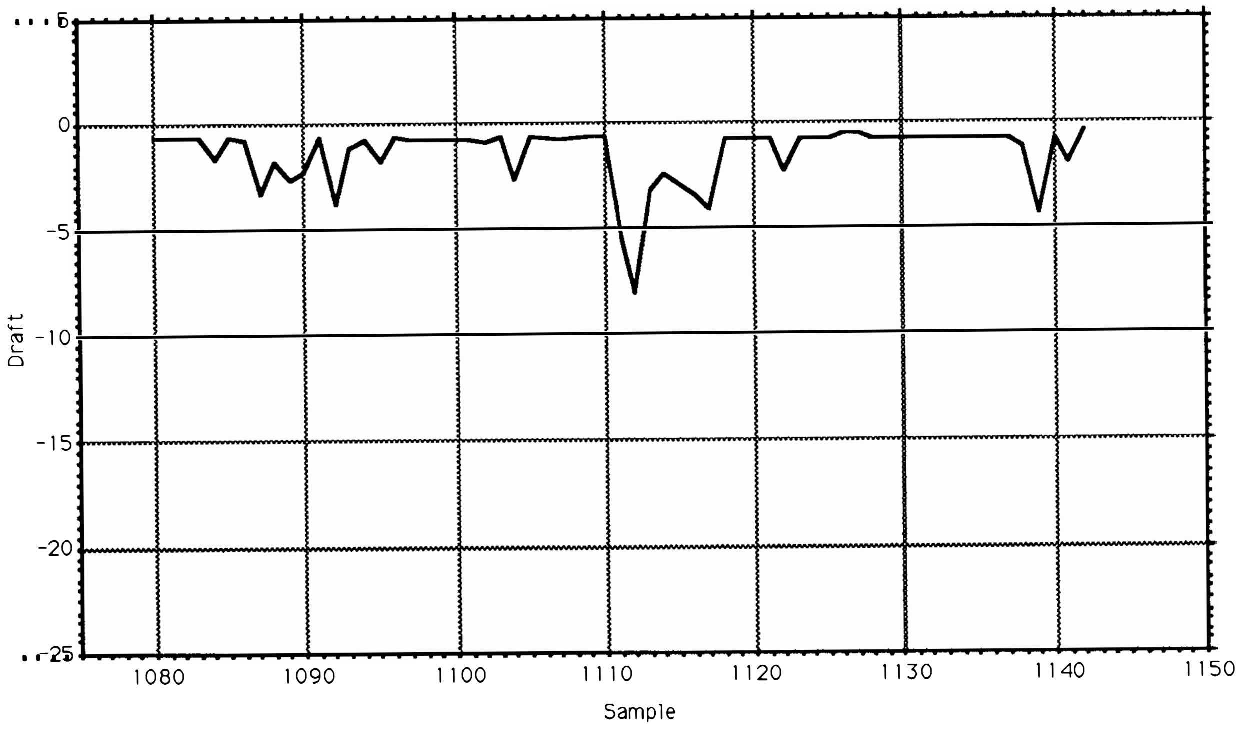
Period 871227 b, uls depth.



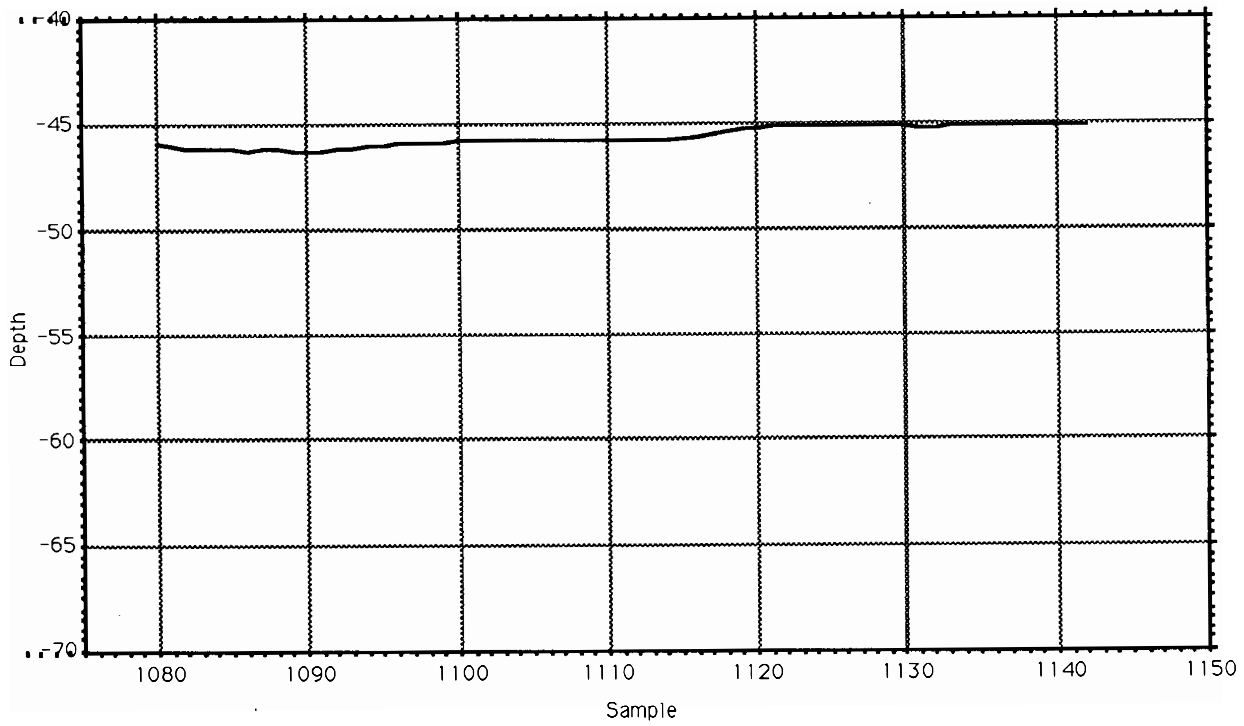
Period 871227 c, draft.



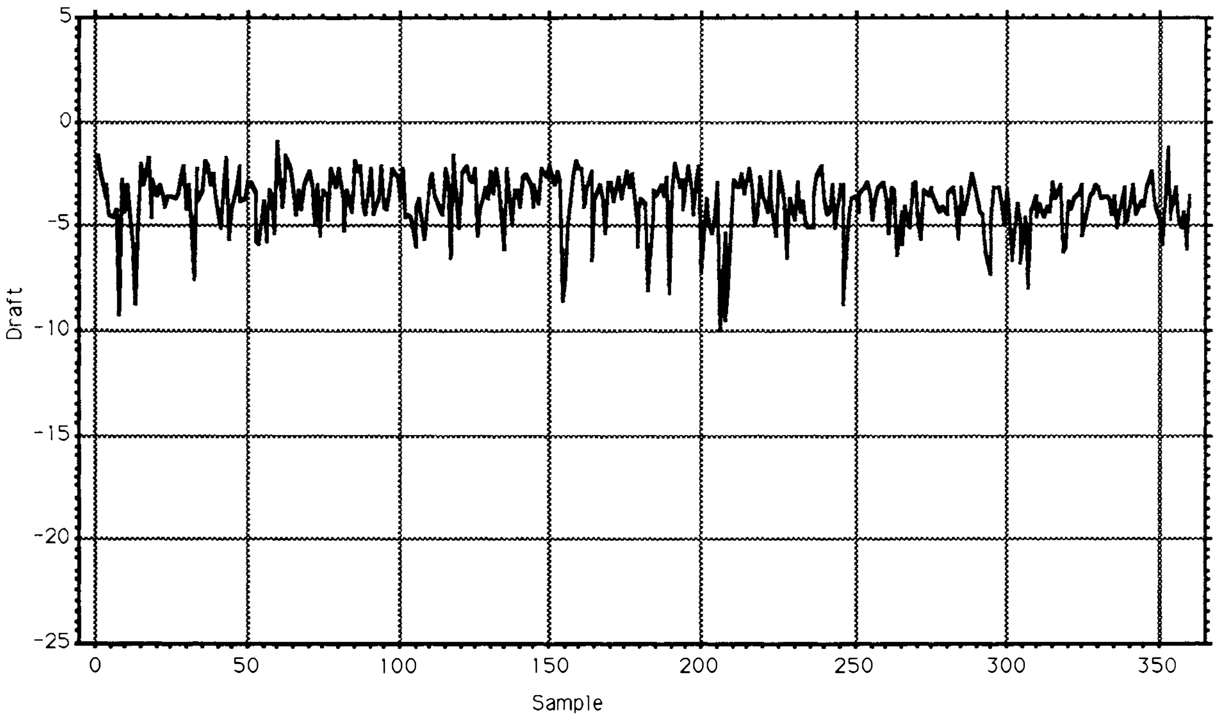
Period 871227 c, uls depth.



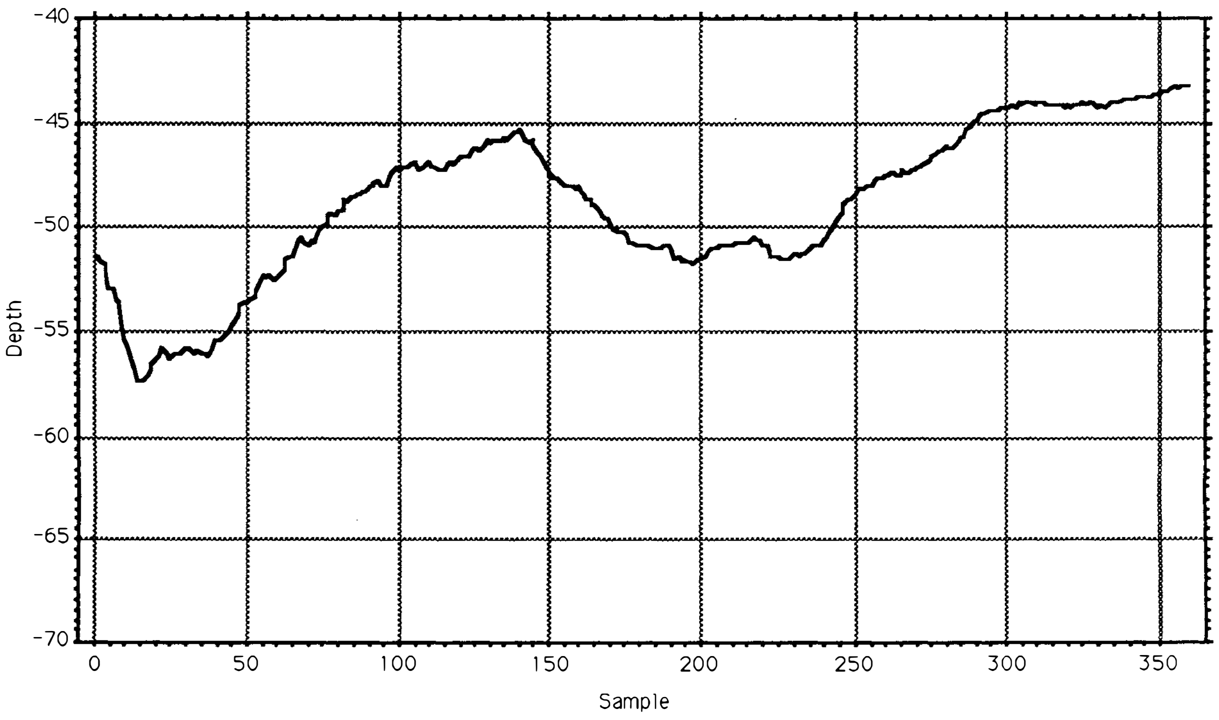
Period 871227 d, draft.



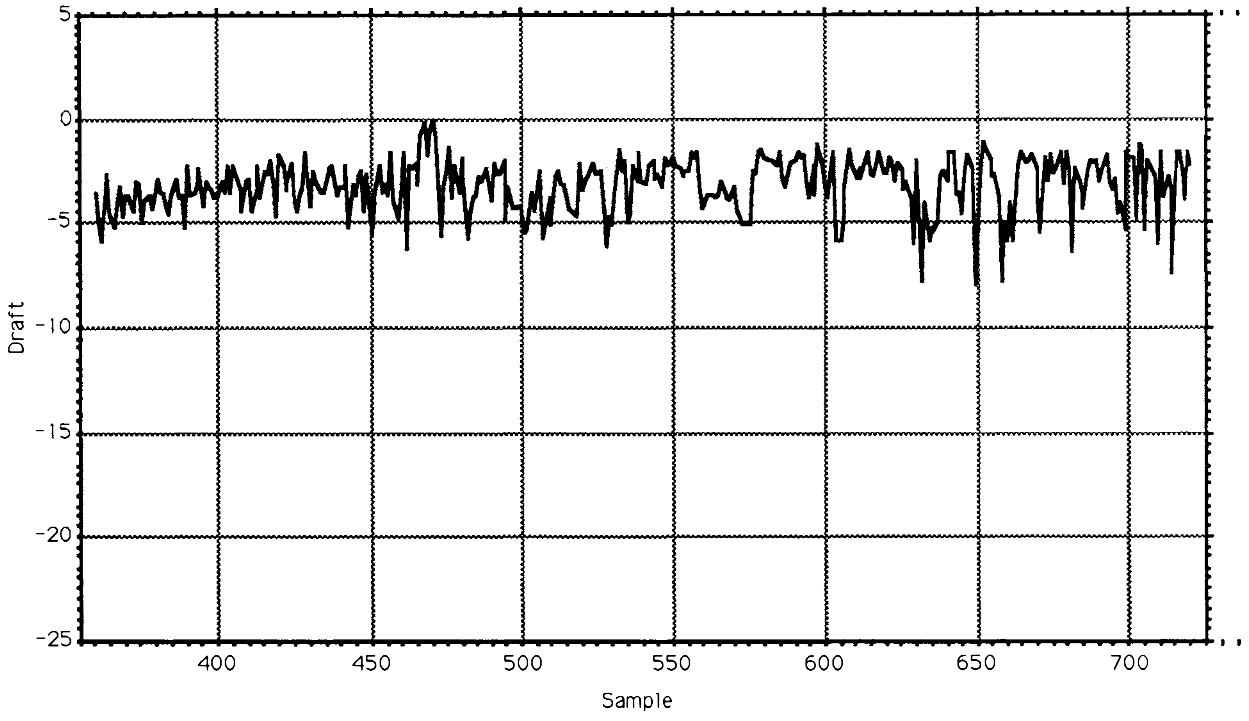
Period 871227 d, uls depth.



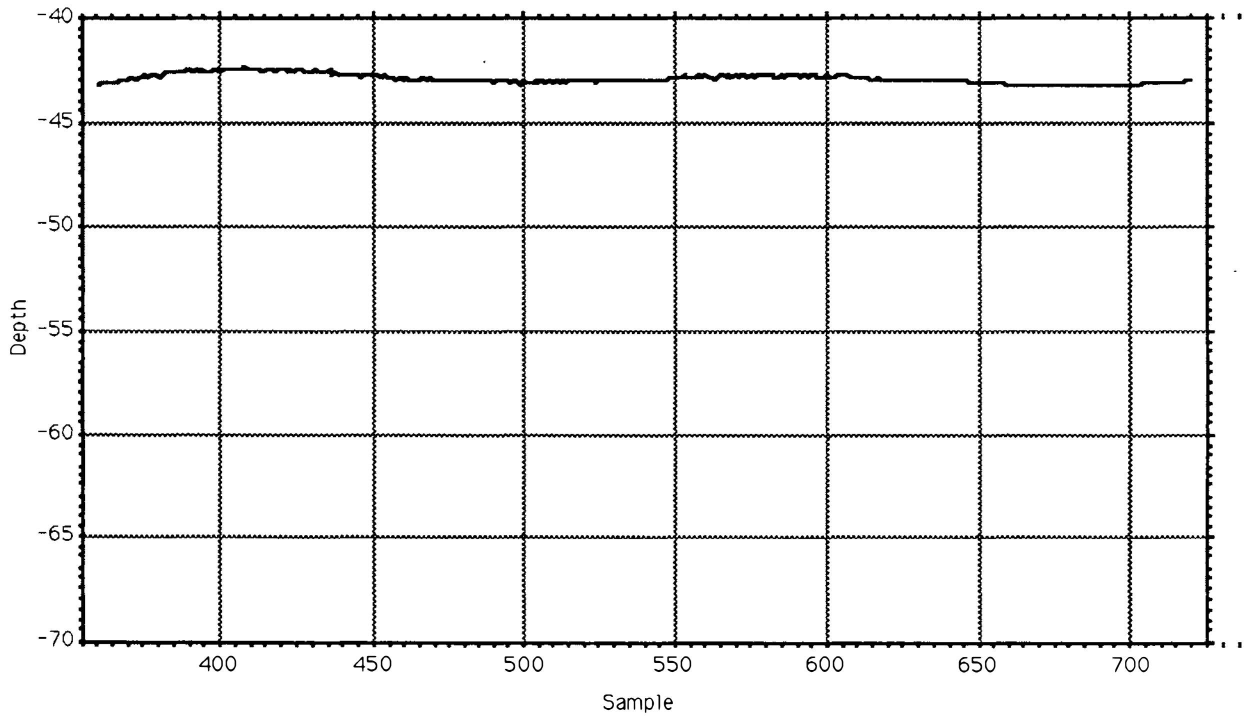
Period 880112 a, draft.



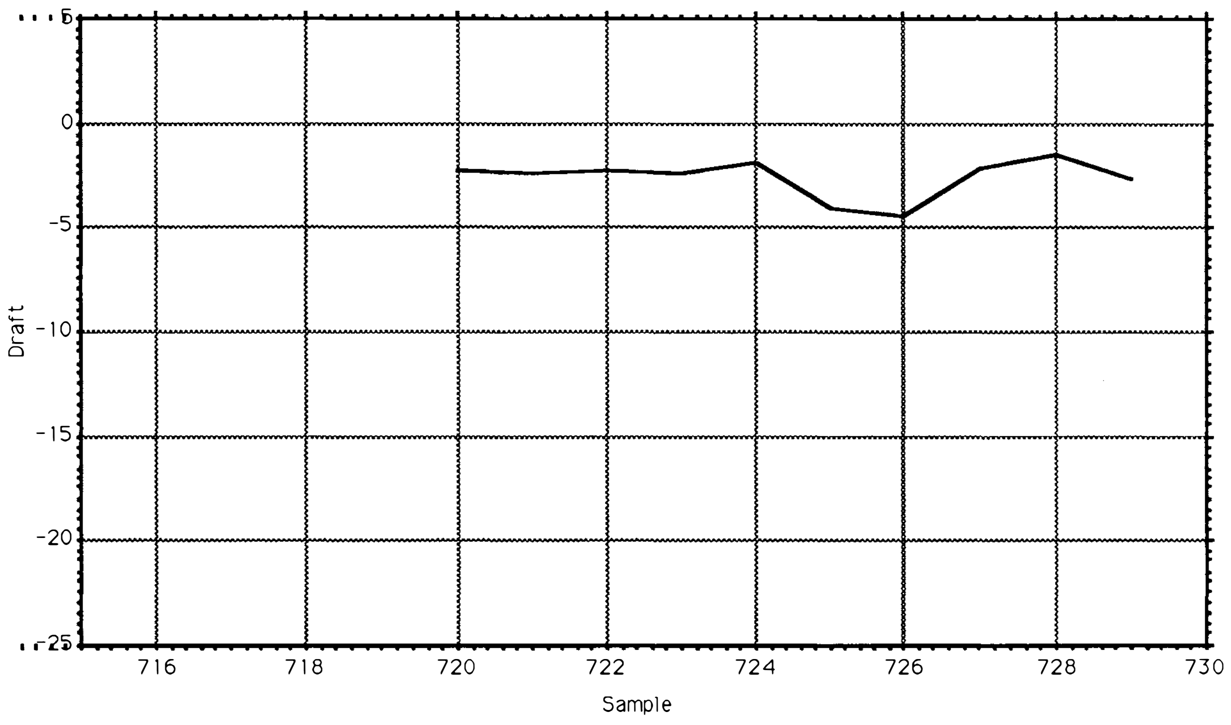
Period 880112 a, uls depth.



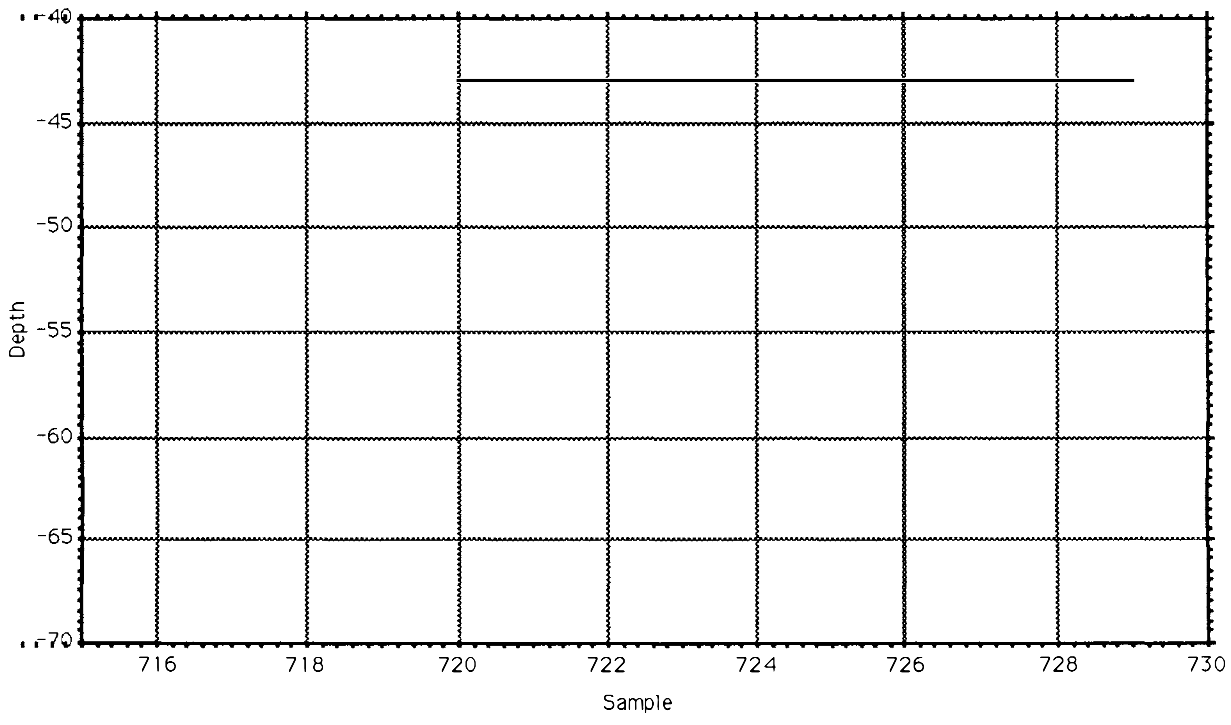
Period 880112 b, draft.



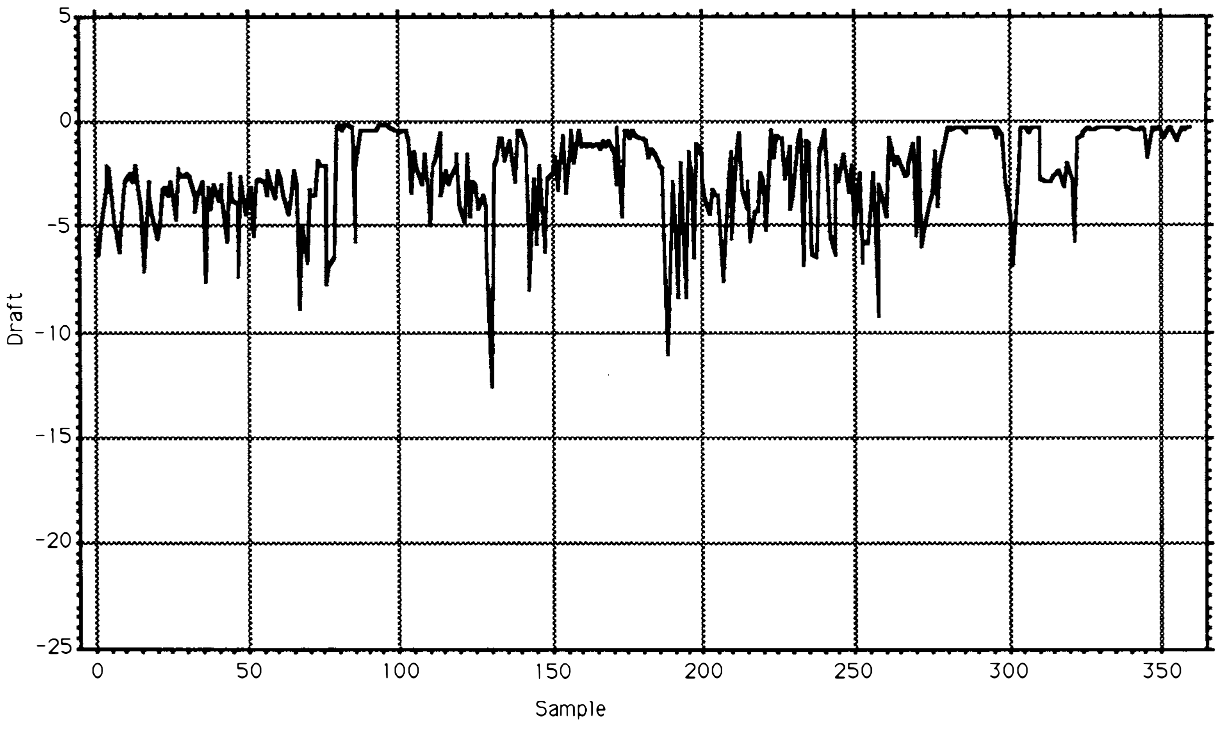
Period 880112 b, uls depth.



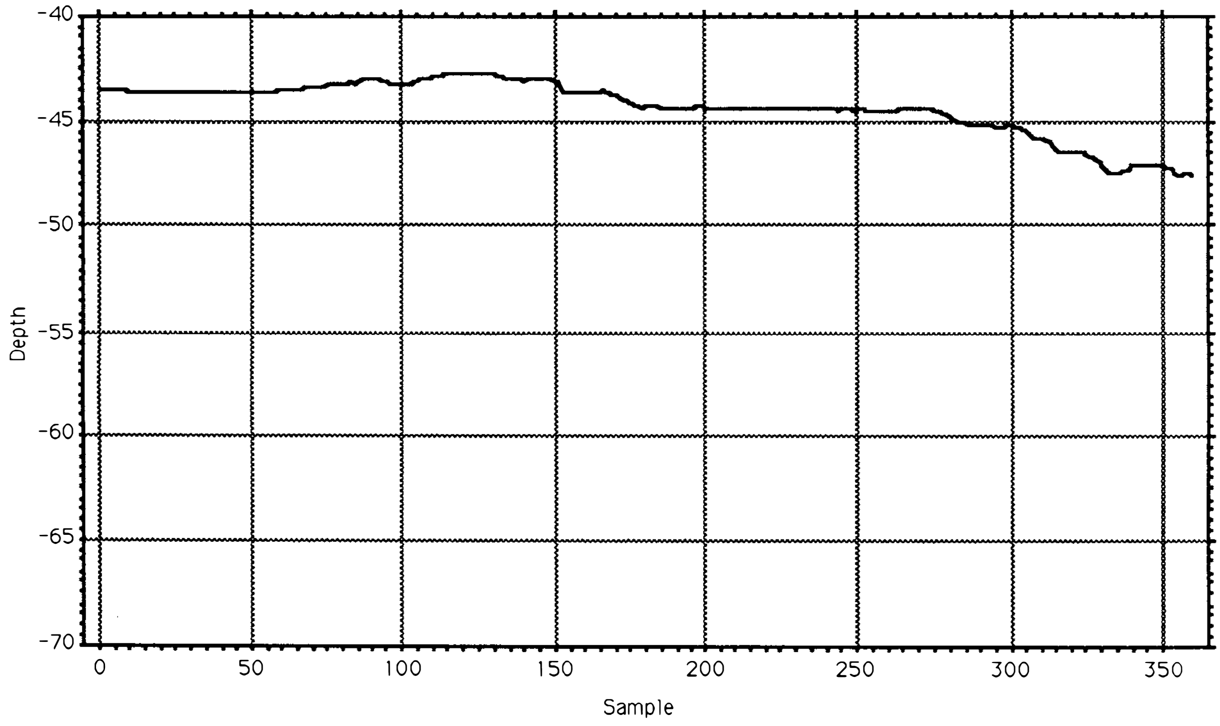
Period 880112 c, draft.



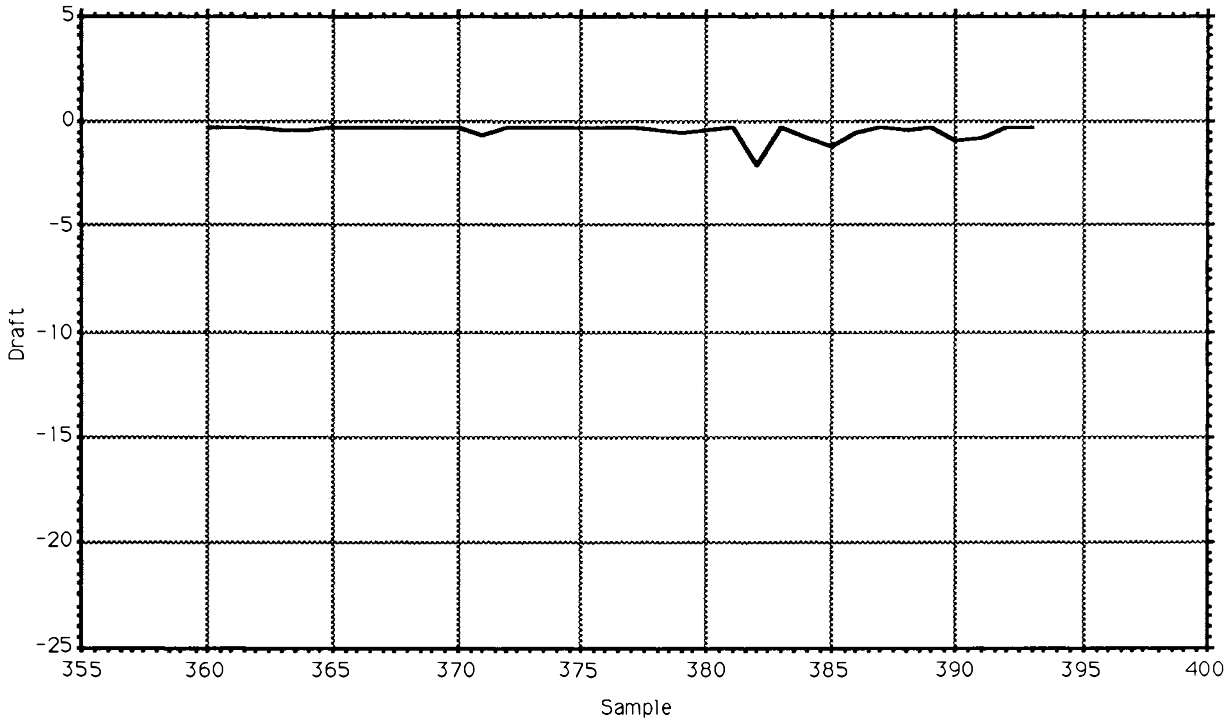
Period 880112 c, uls depth.



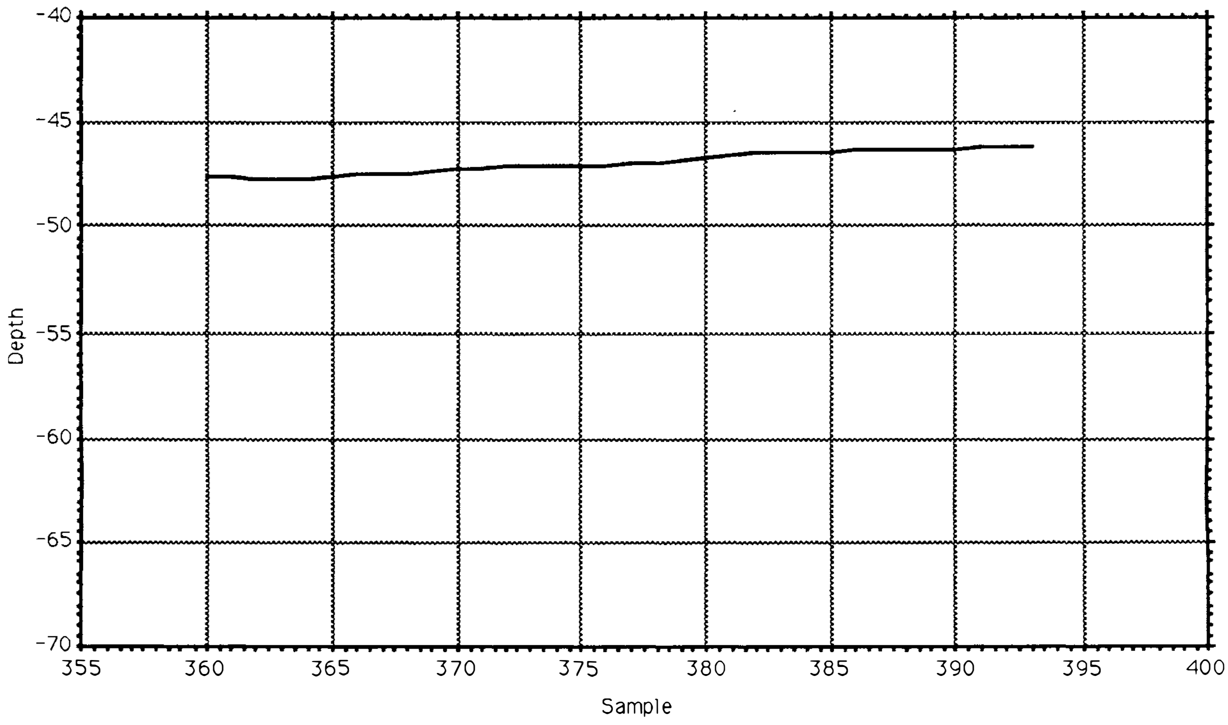
Period 880120 a, draft.



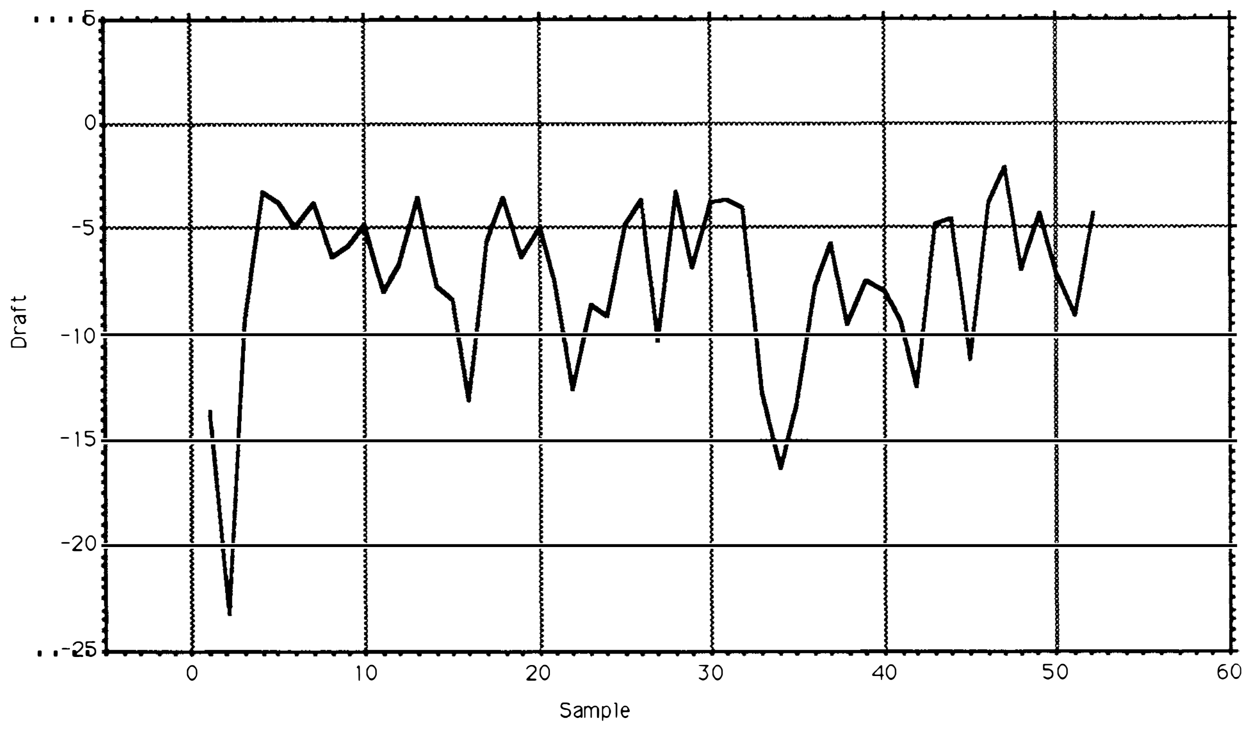
Period 880120 a, uls depth.



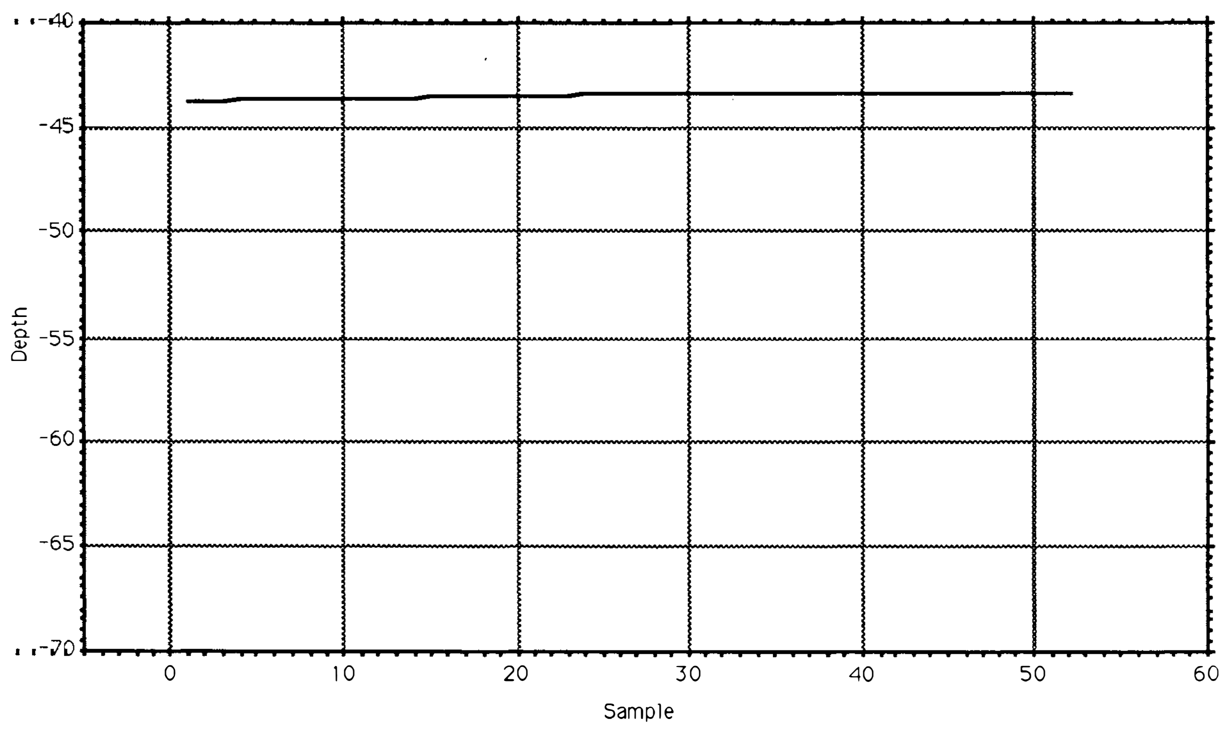
Period 880120 b, draft.



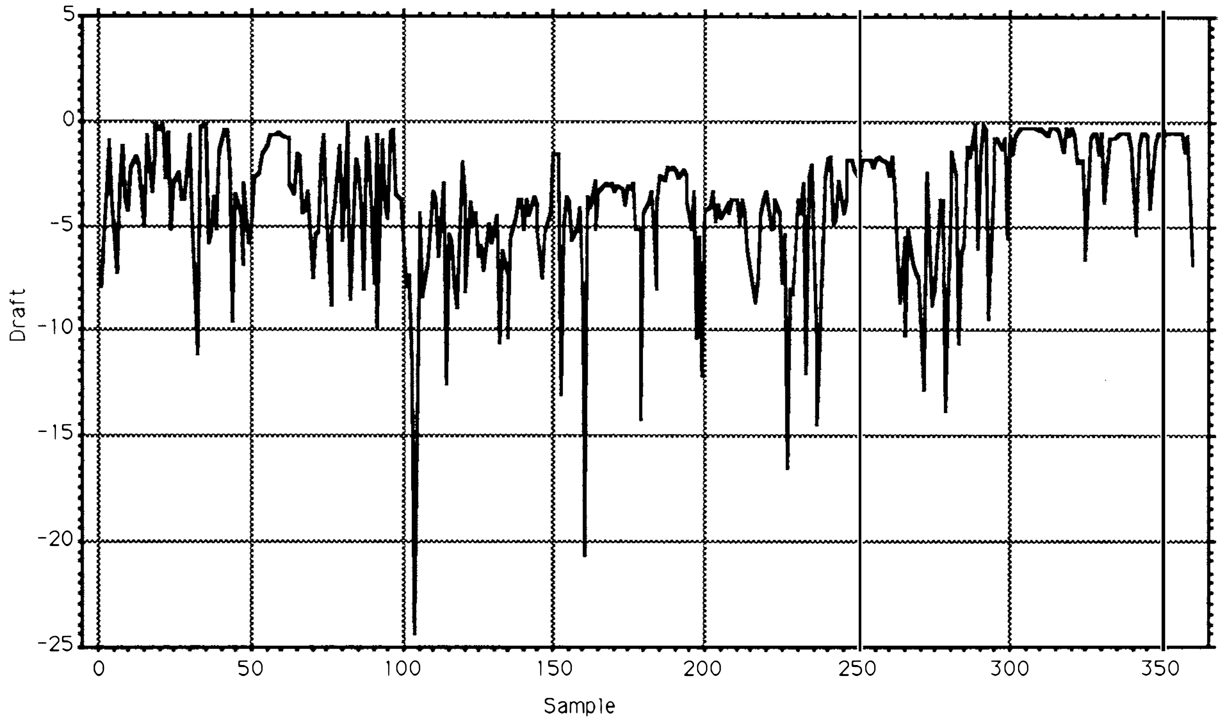
Period 880120 b, uls depth.



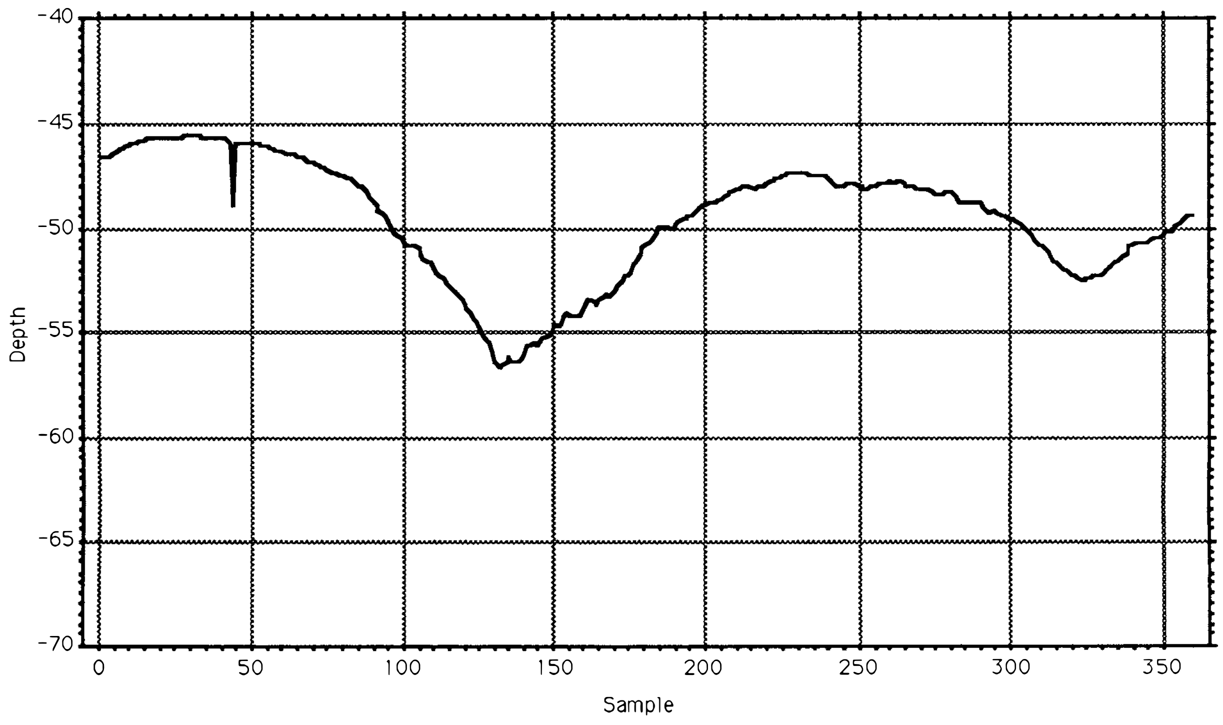
Period 880327 a, draft.



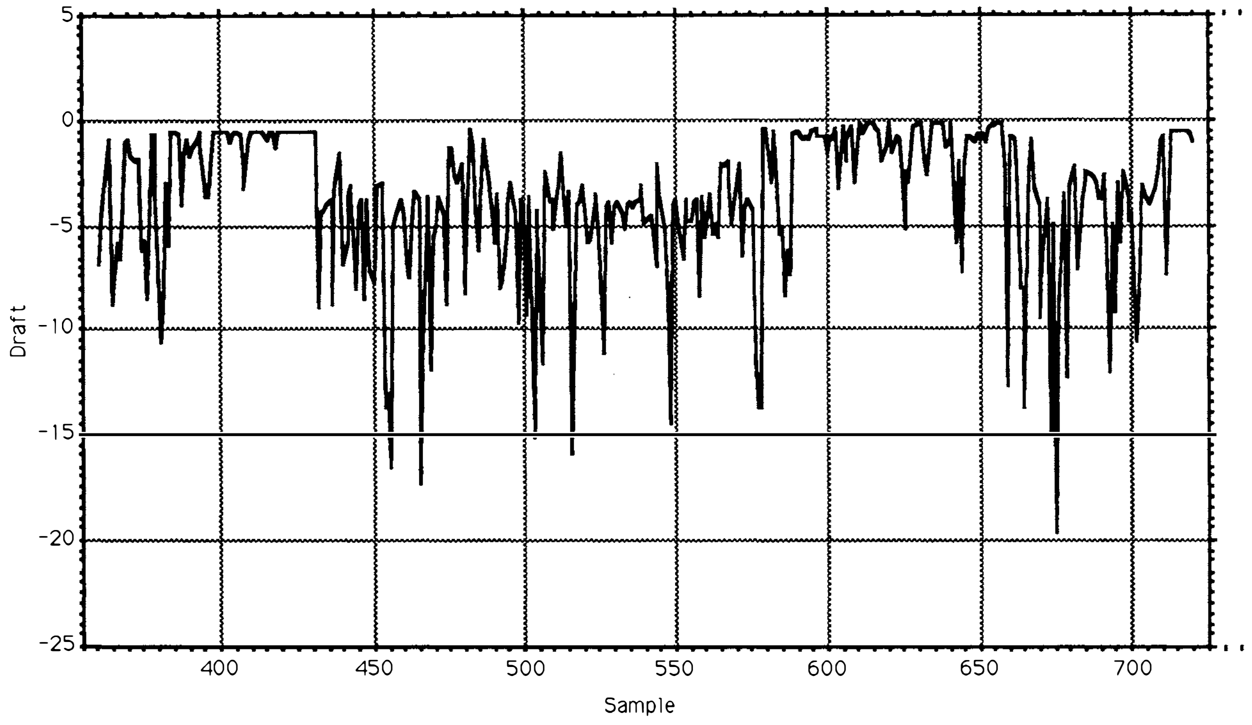
Period 880327 a, uls depth.



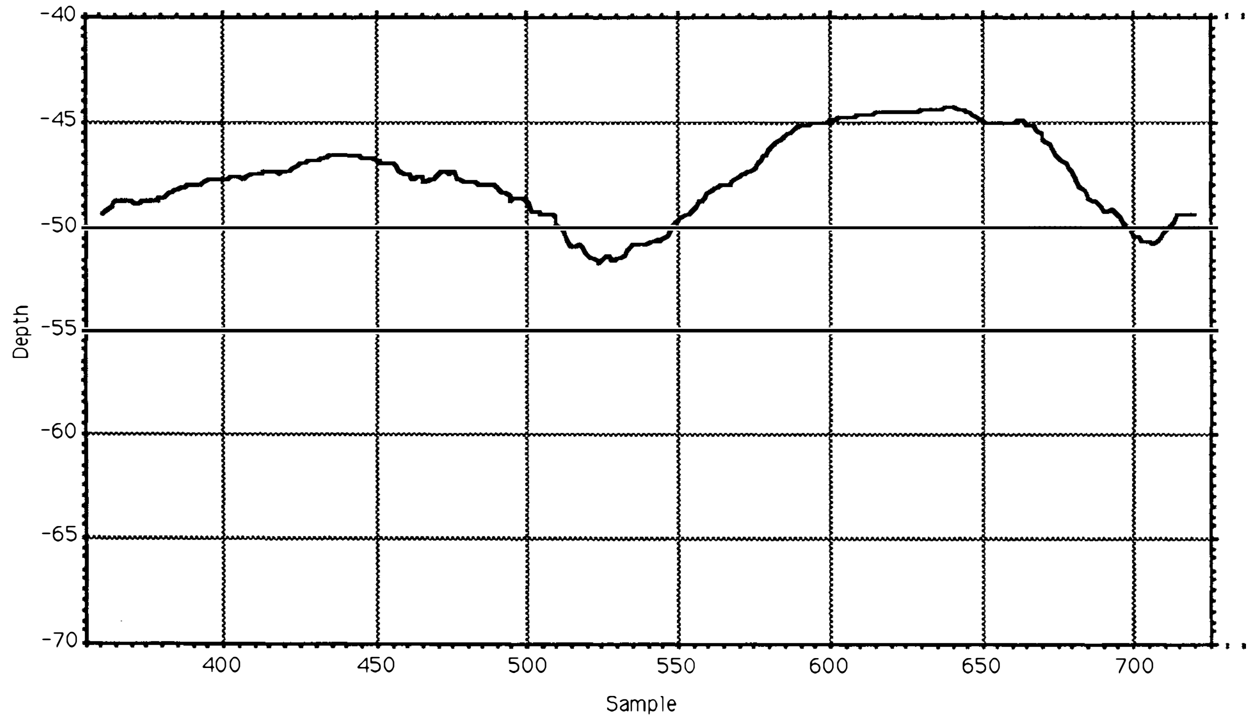
Period 880420 a, draft.



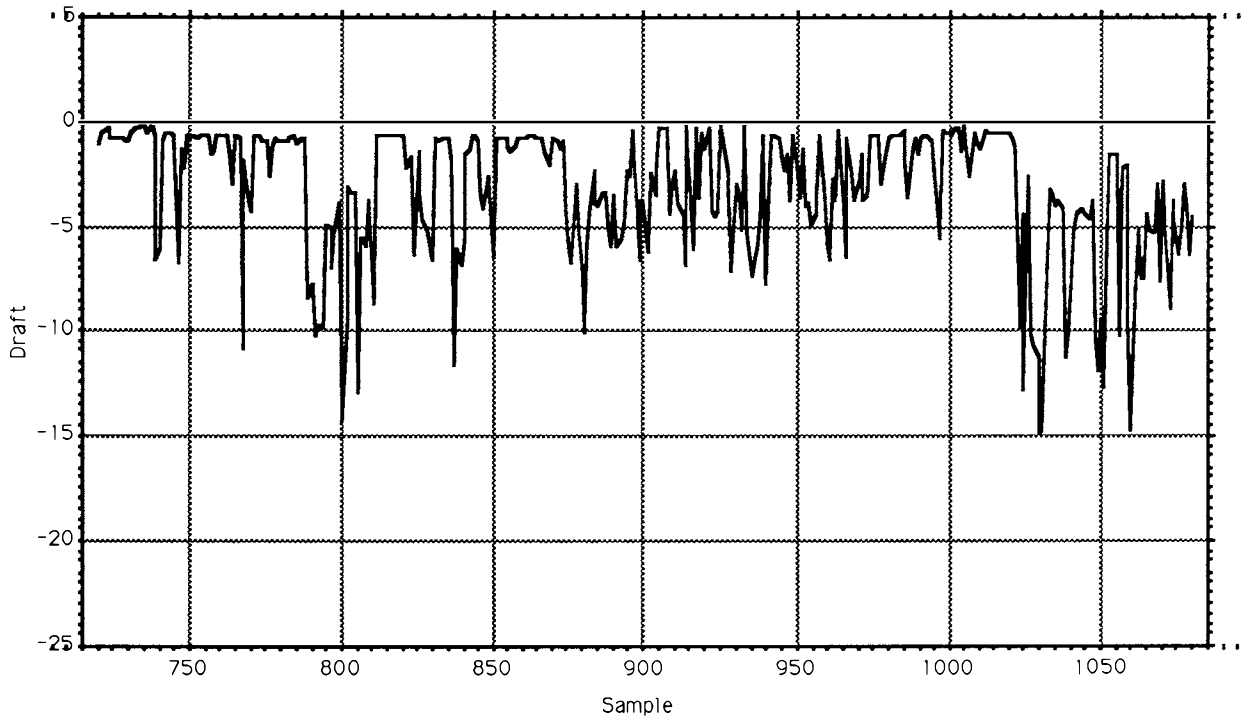
Period 880420 a, uls depth.



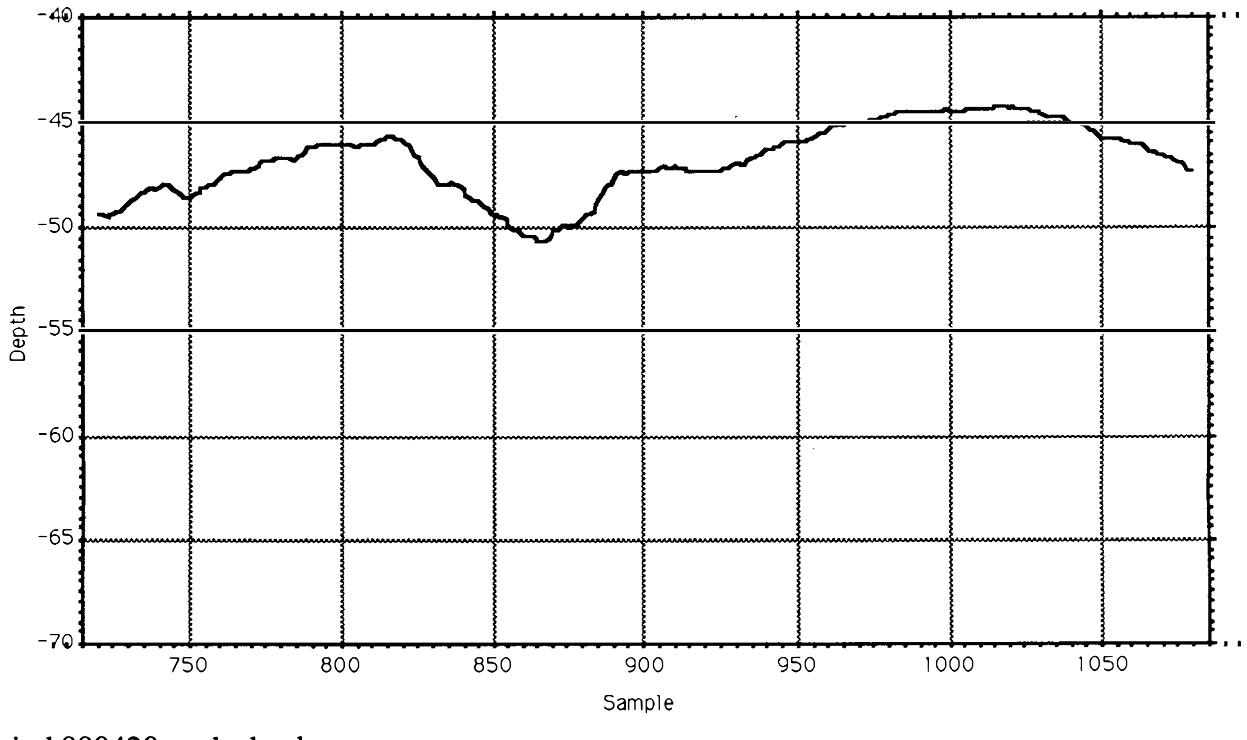
Period 880420 b, draft.



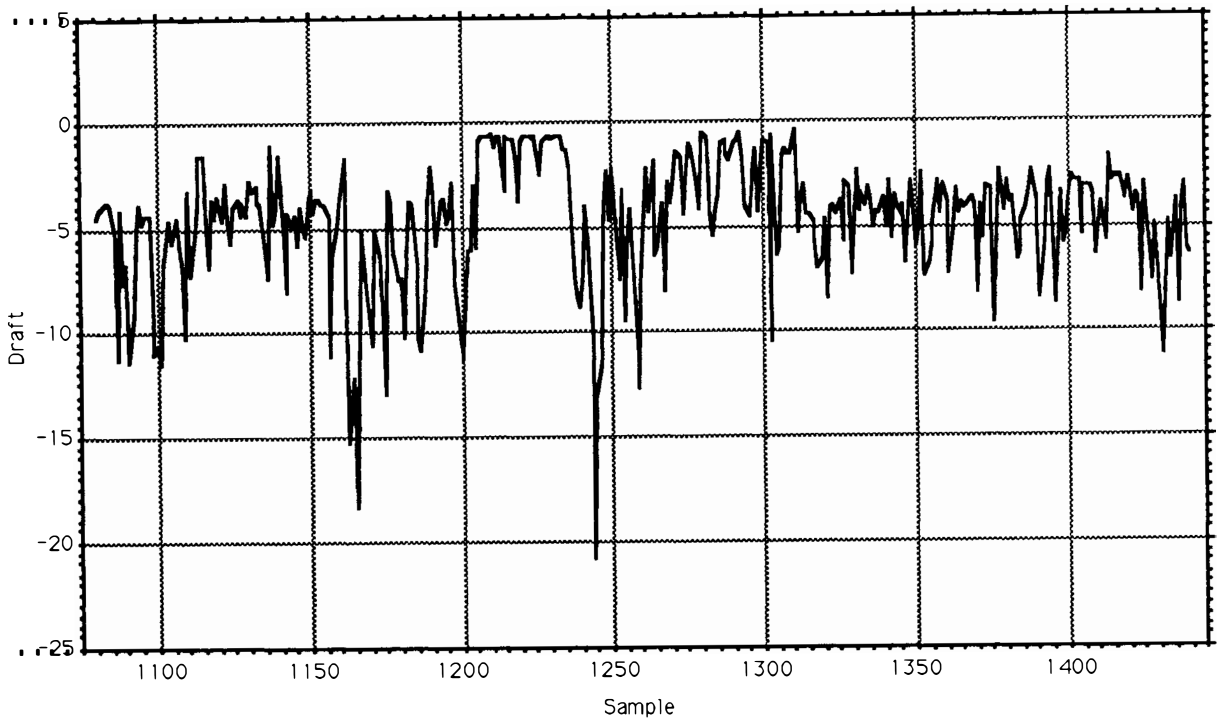
Period 880420 b, uls depth.



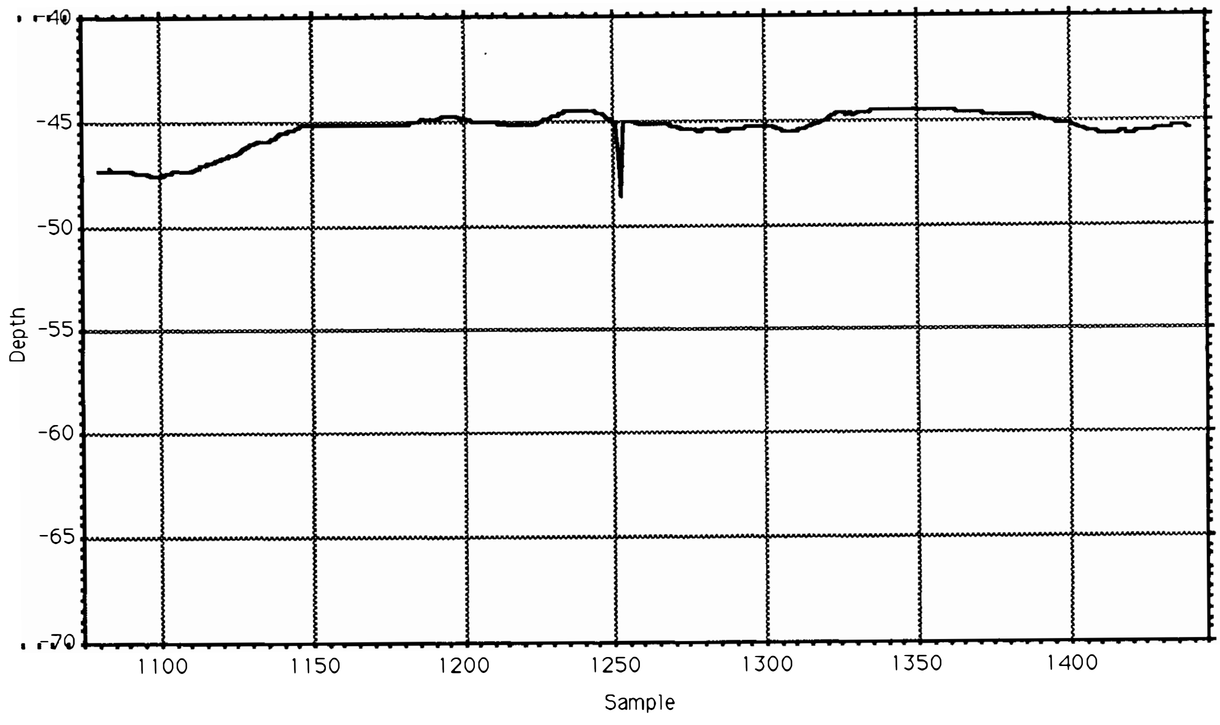
Period 880420 c, draft.



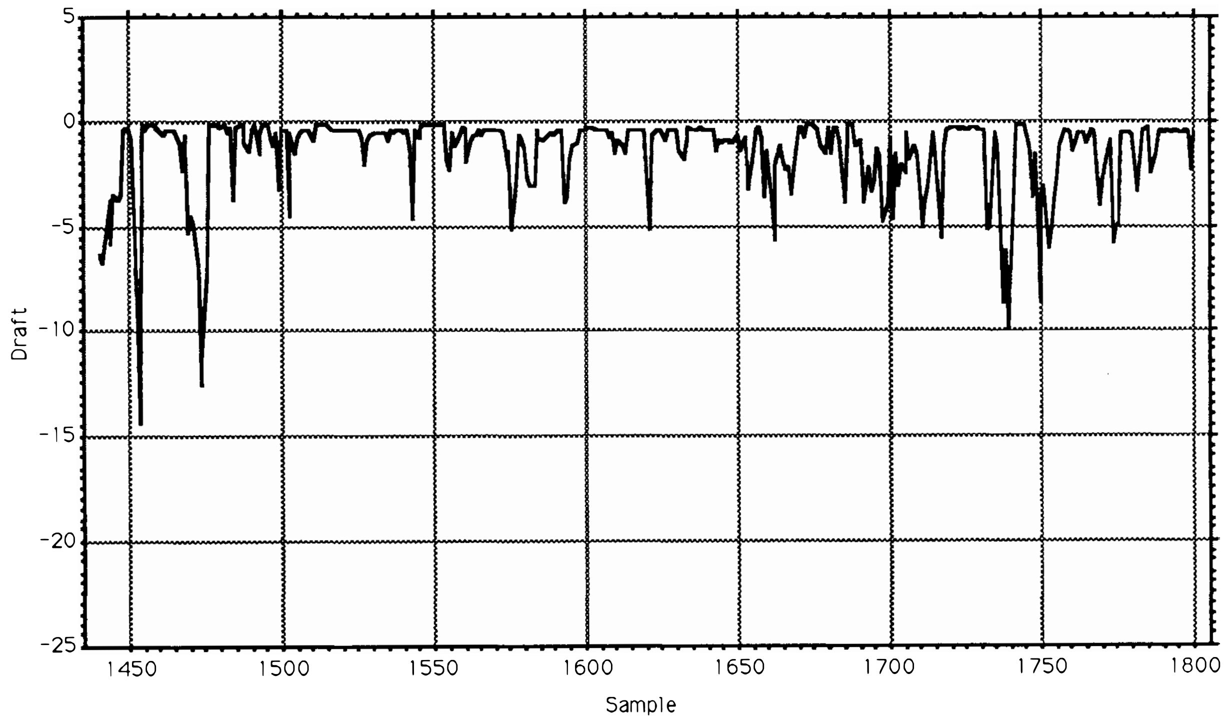
Period 880420 c, uls depth.



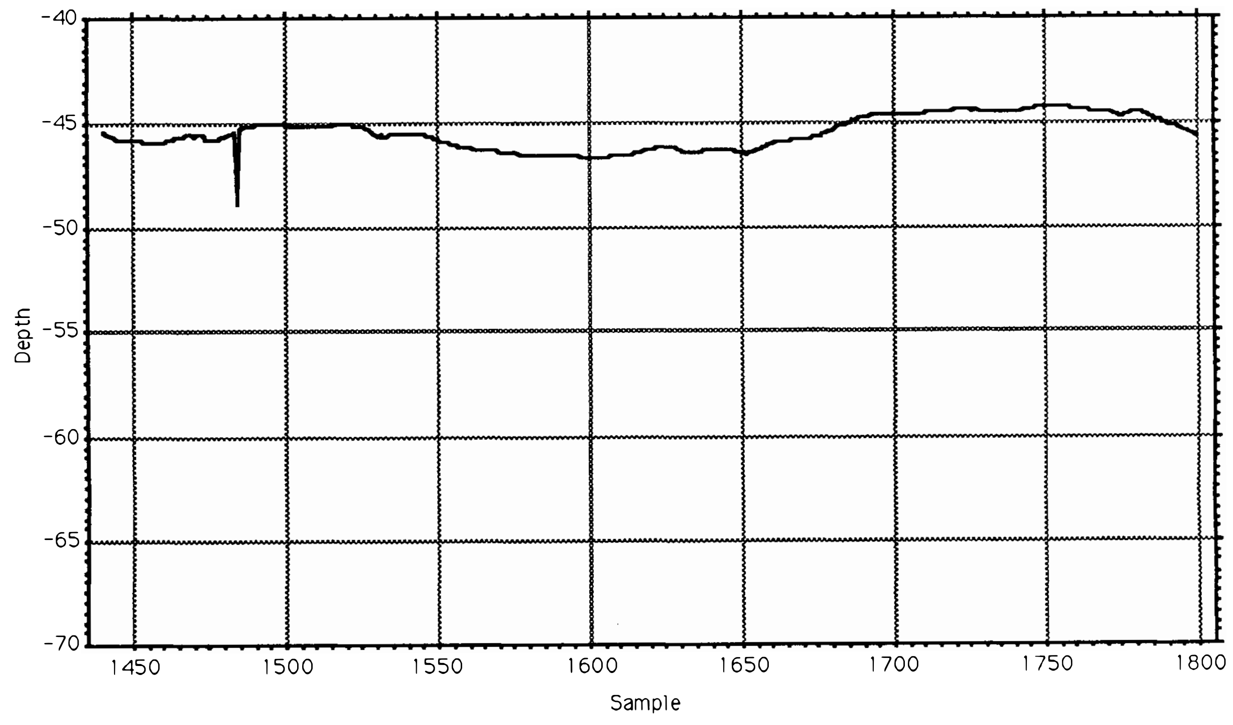
Period 880420 d, draft.



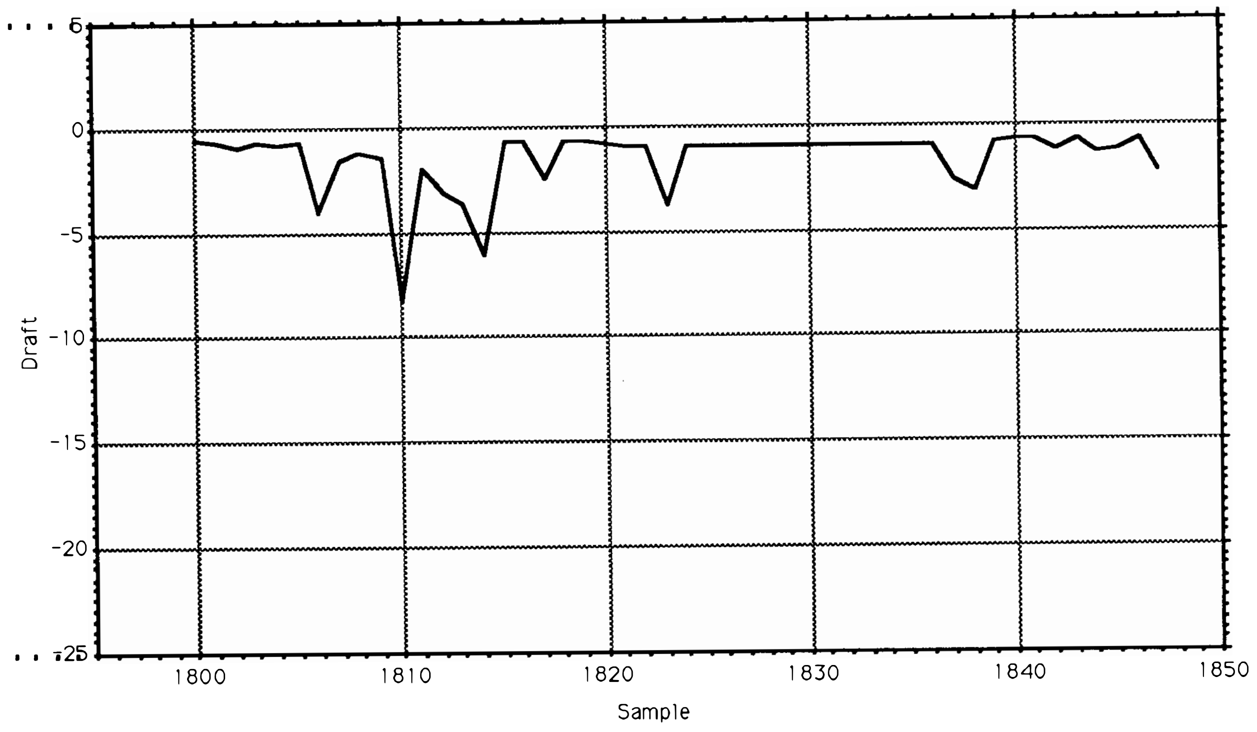
Period 880420 d, uls depth.



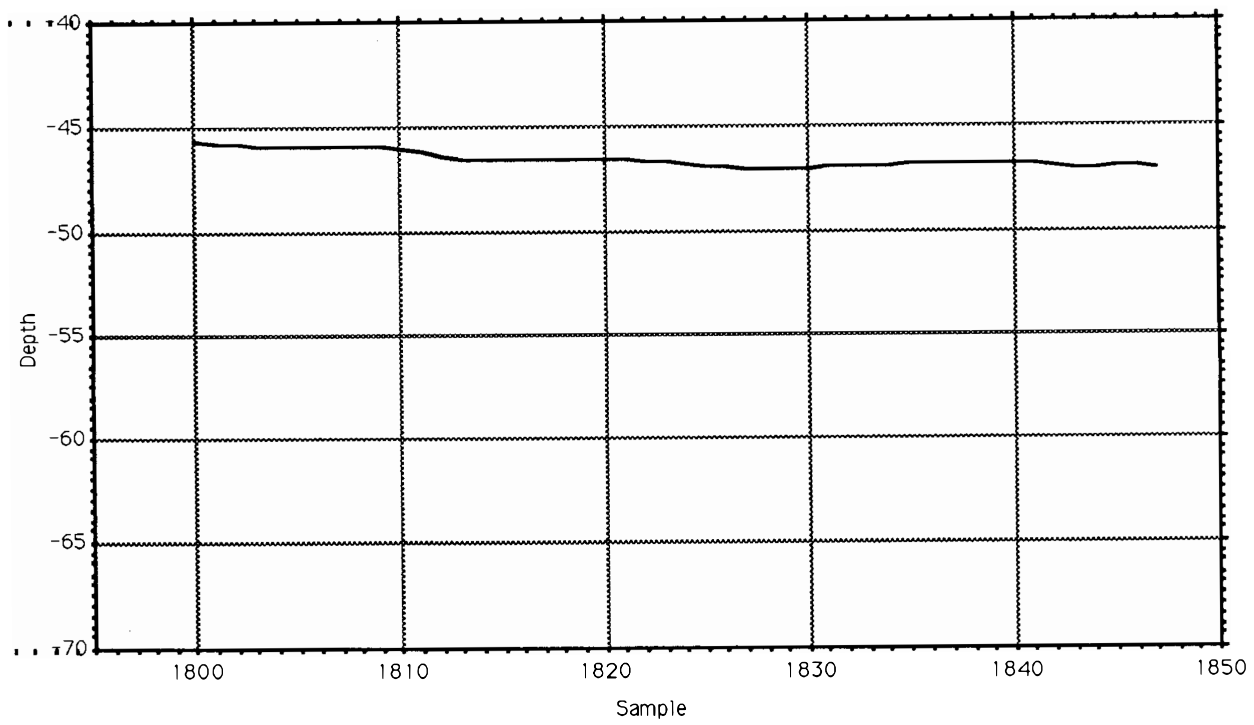
Period 880420 e, draft.



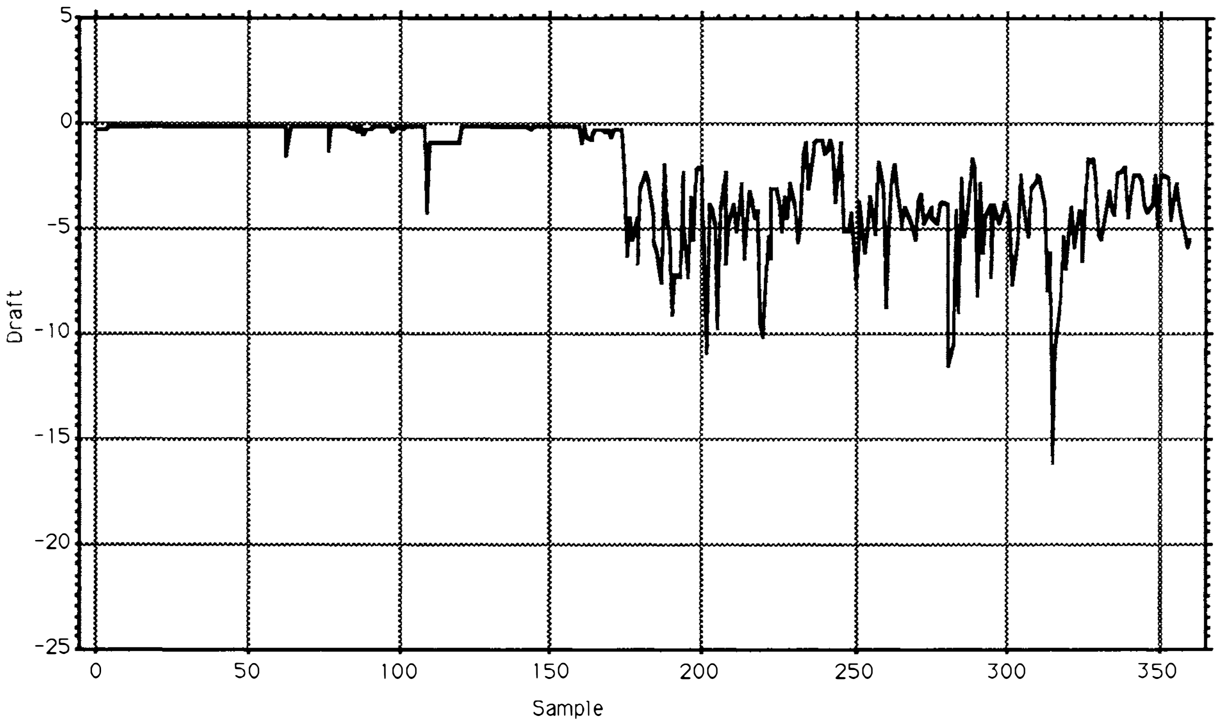
Period 880420 e, uls depth.



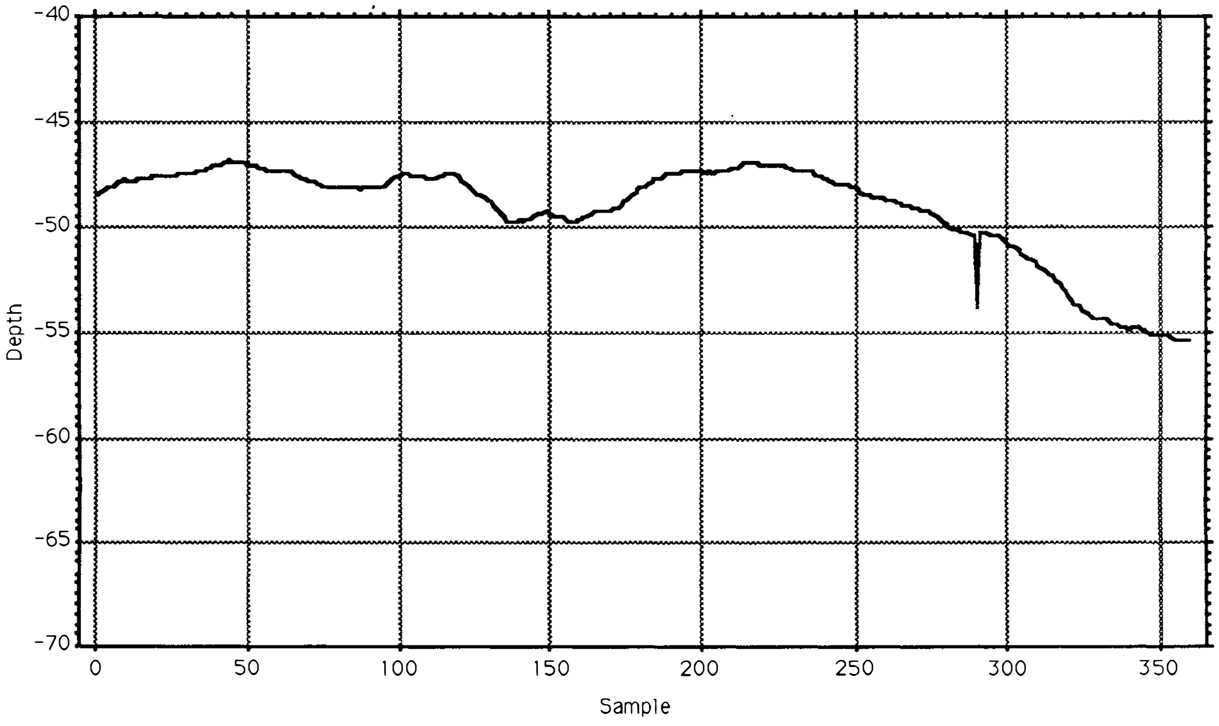
Period 880420 f, draft.



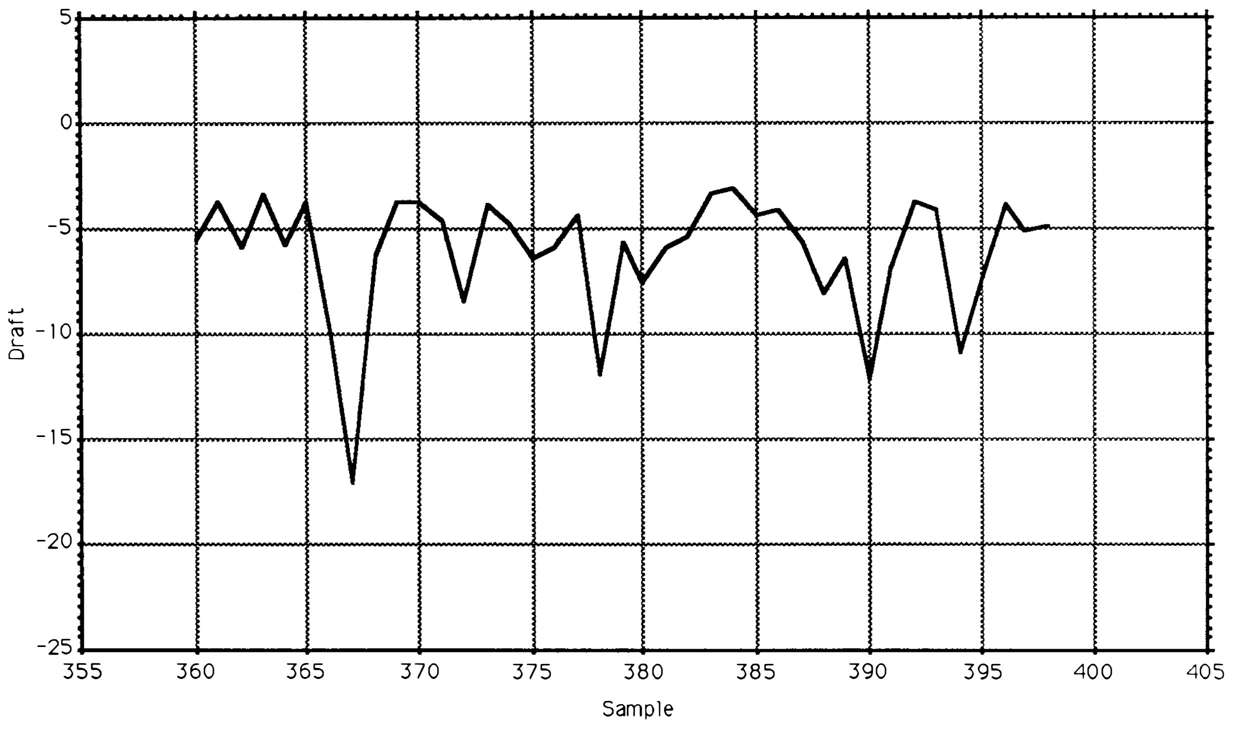
Period 880420 f, uls depth.



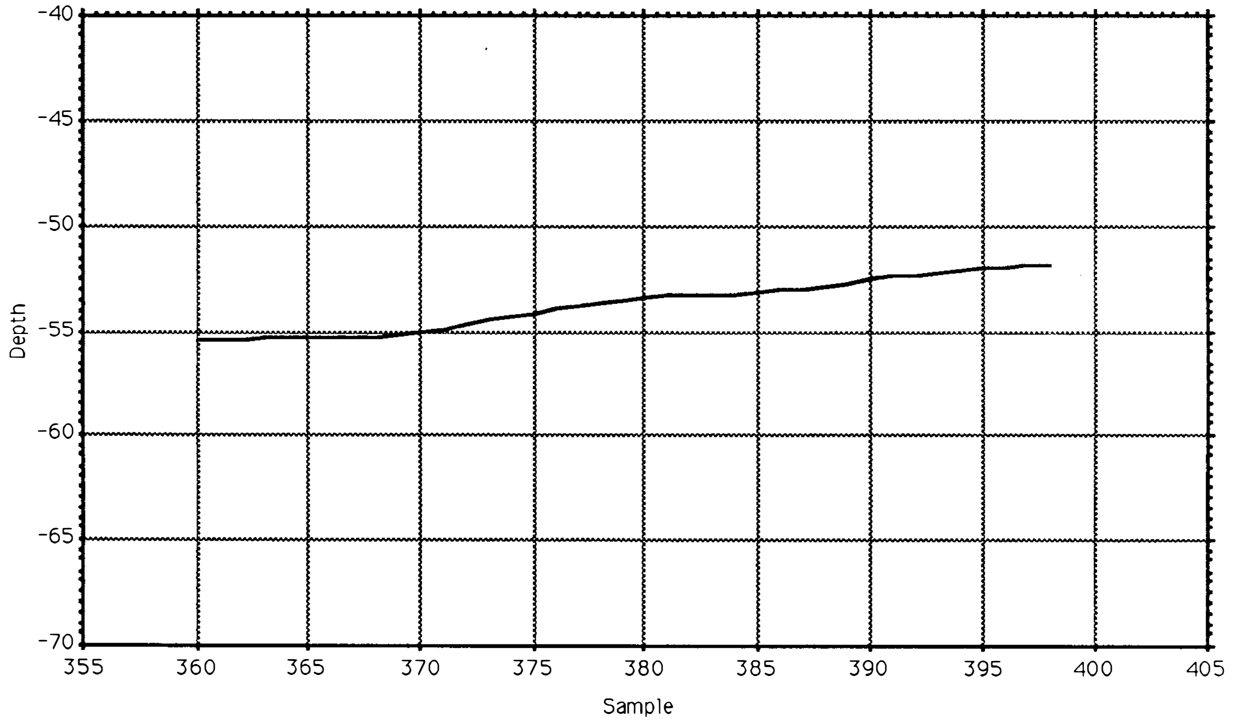
Period 880512 a, draft.



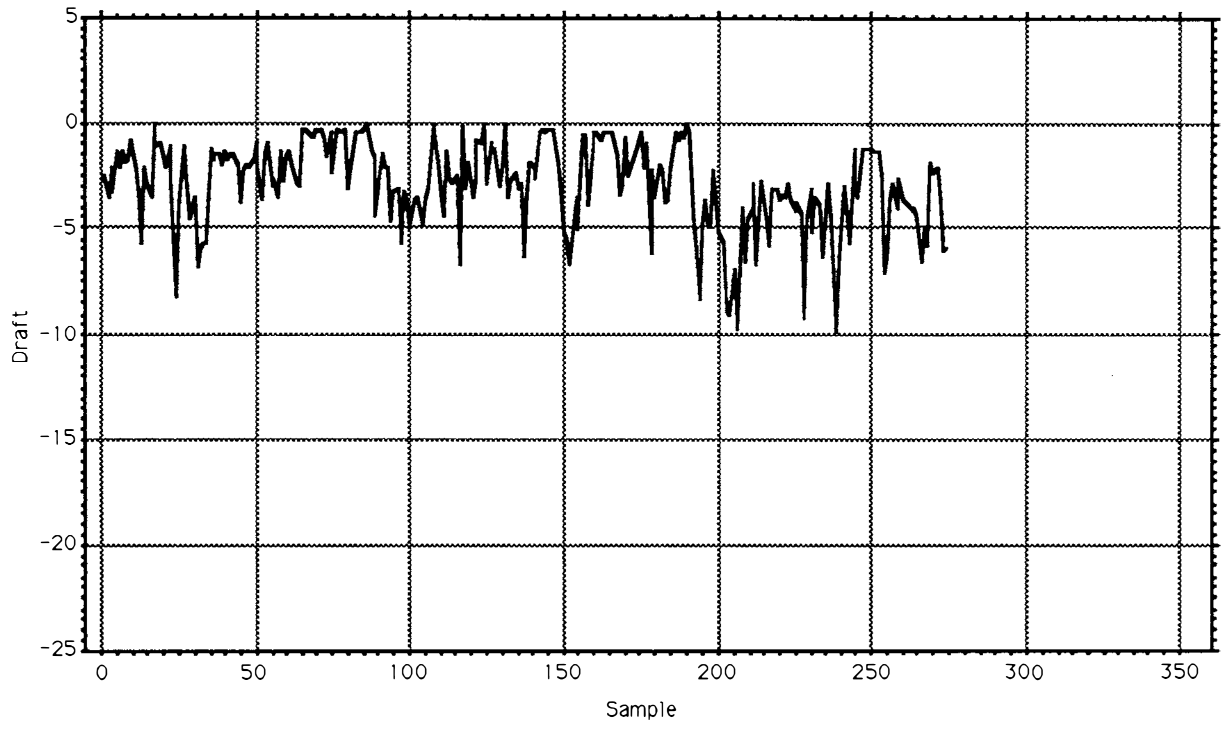
Period 880512 a, uls depth.



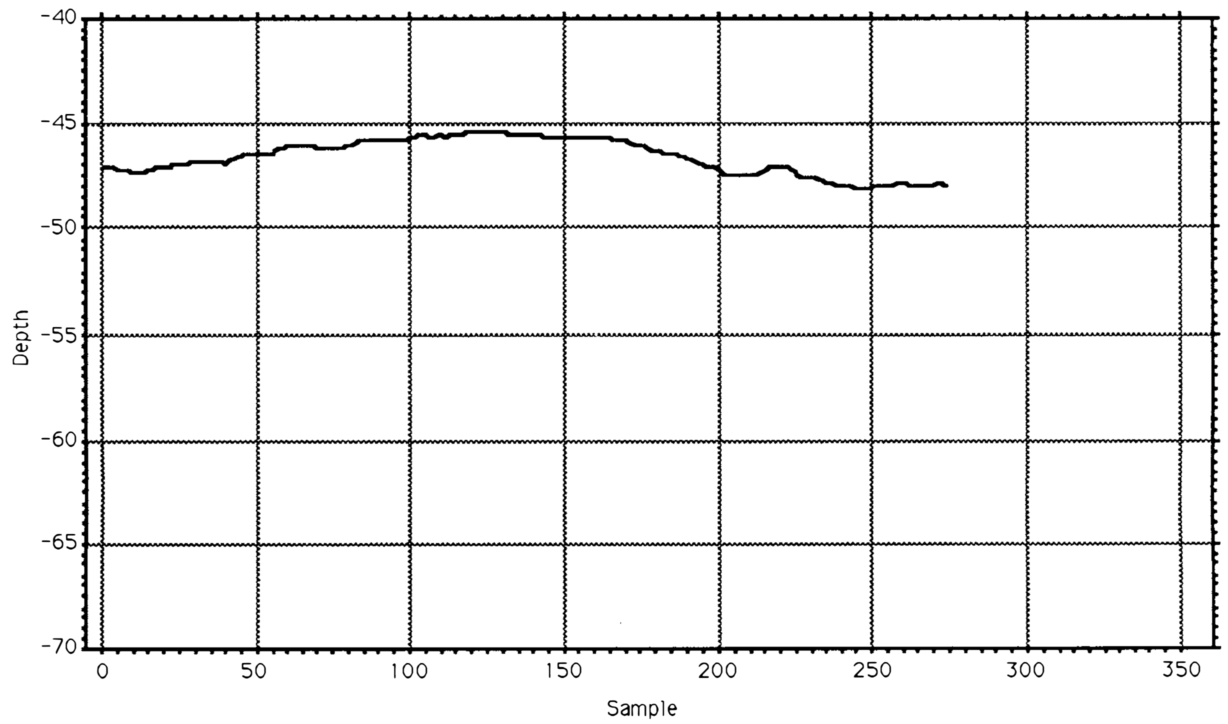
Period 880512 b, draft.



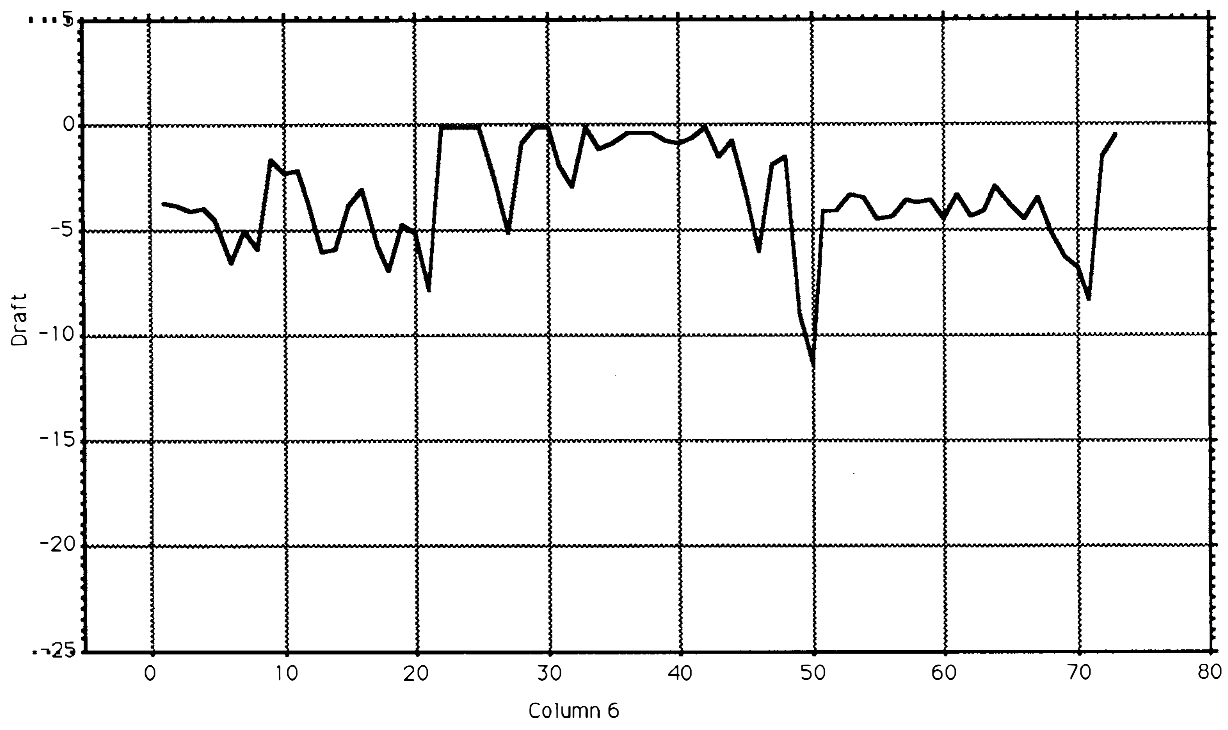
Period 880512 b, uls depth.



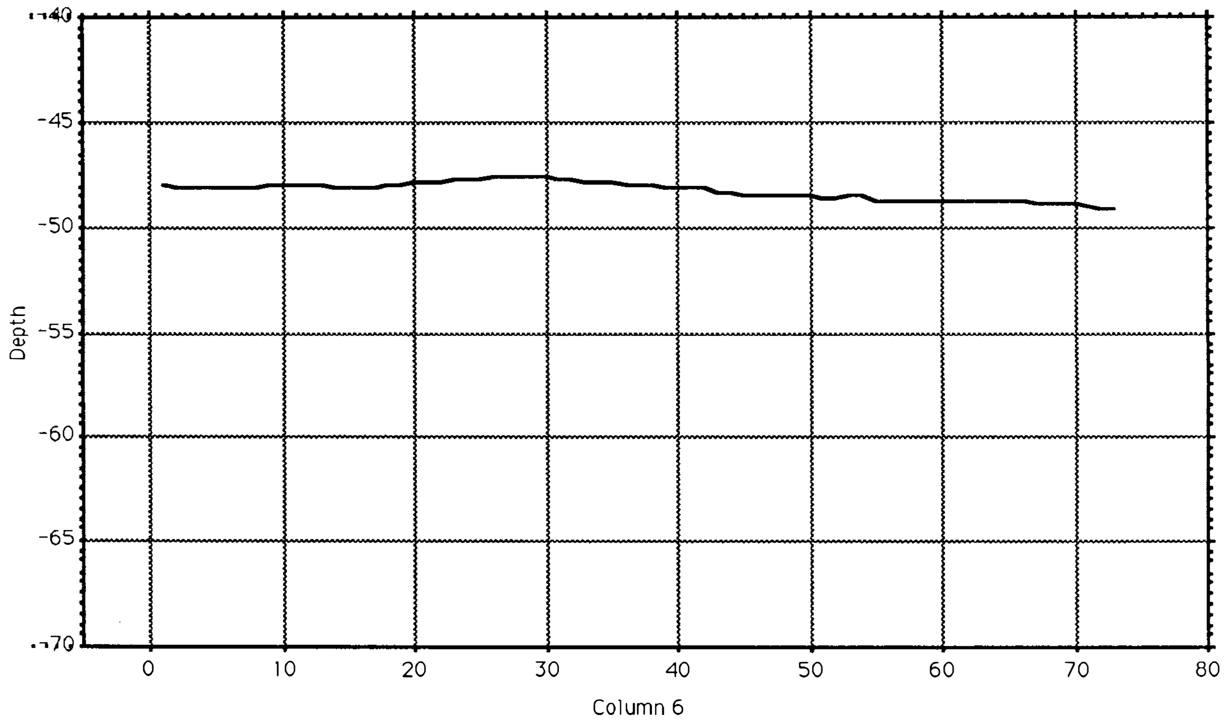
Period 880609 a, draft.



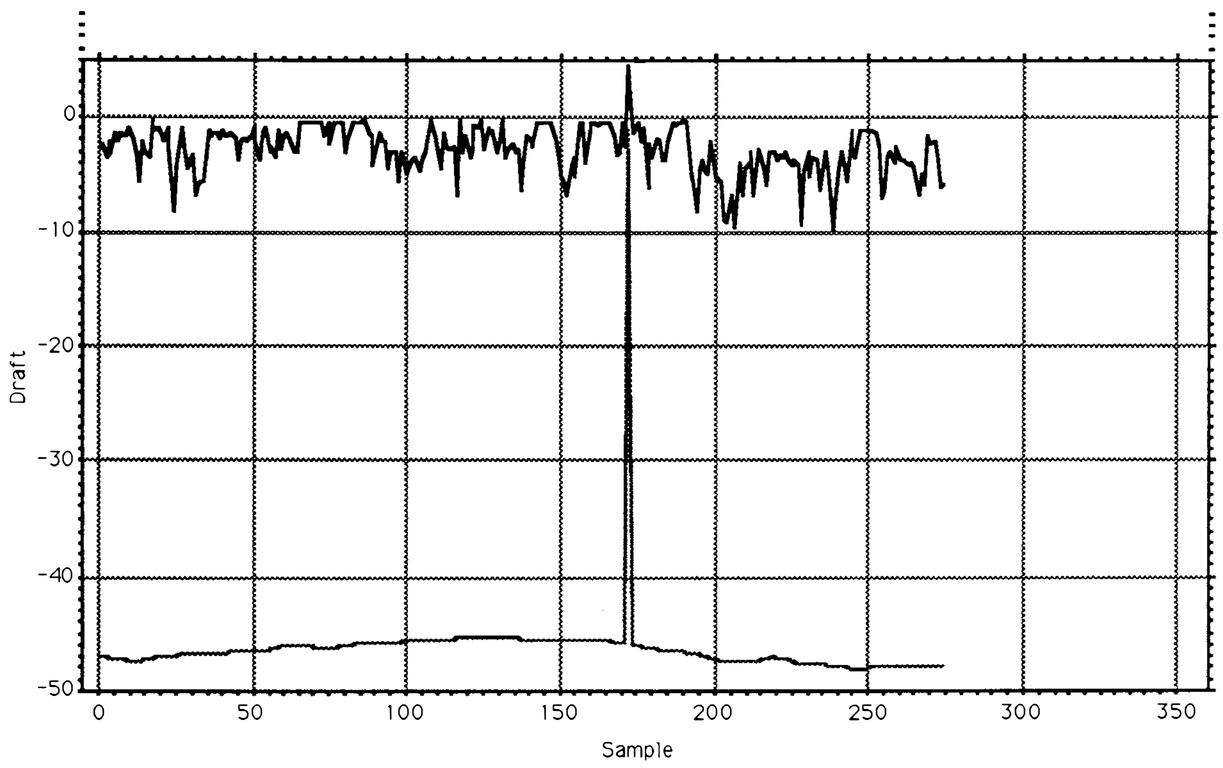
Period 880609 a, uls depth.



Period 880610 a, draft.



Period 880610 a, uls depth.



Period 880609 incuded one gross flier. Those values are not included in the statistics (limit 0.2 for draft).

Descriptive statistics over draft and uls depth during those periods with AVHRR data.

X_1 : Draft

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-1,59	2,88	,146	8,293	-181,149	388

Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
-20,793	-,062	20,731	-616,797	4189,767	0

X_1 : Depth

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-52,659	2,913	,148	8,486	-5,532	388

Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
-58,094	-47,529	10,565	-20431,58	1079184,604	0

Data for the period 881106. Unreasonable values and values above 0.2 m not included.

X_1 : Draft

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-2,551	2,368	,062	5,607	-92,818	1481

Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
-17,381	-,085	17,296	-3778,291	17937,718	0

X_1 : Depth

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-45,791	,921	,024	,849	-2,012	1481

Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
-47,961	-44,083	3,878	-67816,434	3106636,42	0

Data for the period 881116. Unreasonable values and values above 0.2 m not included.

X_1 : Draft

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-2,185	2,563	,08	6,569	-117,298	1016

Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
-14,855	-,094	14,761	-2220,071	11519,043	0

X_1 : Depth

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-50,936	3,643	,114	13,269	-7,151	1016

Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
-60,392	-44,704	15,688	-51750,919	2649449,538	0

Data for the period 881120. Unreasonable values and values above 0.2 m not included.

X_1 : Draft

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-2,372	2,671	,096	7,135	-112,621	768
Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
-14,827	-,086	14,741	-1821,525	9792,719	0

X_1 : Depth

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-44,971	,968	,035	,938	-2,153	768
Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
-47,598	-43,469	4,129	-34537,892	1553930,125	0

Data for the period 881204. Unreasonable values and values above 0.2 m not included.

X_1 : Draft

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-4,823	3,559	,202	12,666	-73,786	309
Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
-19,301	-,21	19,091	-1490,426	11090,099	0

X_1 : Depth

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-45,826	,621	,035	,385	-1,355	309
Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
-46,96	-44,906	2,054	-14160,135	649016,501	0

Data for the period 881215. Unreasonable values and values above 0.2 m not included.

X_1 : Draft

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-4,021	3,537	,105	12,512	-87,967	1131
Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
-23,163	-,144	23,019	-4547,81	32425,36	0

X_1 : Depth

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-50,401	5,617	,167	31,547	-11,144	1131
Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
-68,895	-44,701	24,194	-57003,74	2908704,445	0

Data for the period 881227. Unreasonable values and values above 0.2 m not included.

X_1 : Draft

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-3,55	1,396	,052	1,95	-39,342	727

Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
-9,963	-,064	9,899	-2580,508	10575,366	0

X_1 : Depth

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-45,868	3,94	,146	15,525	-8,59	727

Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
-57,409	-42,436	14,973	-33345,862	1540770,894	0

Data for the period 880112. Unreasonable values and values above 0.2 m not included.

X_1 : Draft

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-2,335	2,105	,107	4,432	-90,163	390

Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
-12,593	-,171	12,422	-910,653	3850,546	0

X_1 : Depth

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-44,52	1,426	,072	2,035	-3,204	390

Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
-47,699	-42,659	5,04	-17362,836	773786,53	0

Data for the period 880120. Unreasonable values and values above 0.2 m not included.

X_1 : Draft

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-7,399	4,014	,557	16,111	-54,248	52

Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
-23,228	-2,078	21,15	-384,75	3668,42	0

X_1 : Depth

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-43,462	,128	,018	,016	-,294	52

Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
-43,781	-43,324	,457	-2260,002	98224,082	0

Data for the period 880327. Unreasonable values and values above 0.2 m not included.

X_1 : Draft

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-3,374	3,168	,074	10,037	-93,905	1835
Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
-24,313	-,144	24,169	-6190,904	39294,91	0

X_1 : Depth

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-46,979	2,36	,055	5,571	-5,024	1835
Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
-56,544	-44,261	12,283	-86206,524	4060115,846	0

Data for the period 880420. Unreasonable values and values above 0.2 m not included.

X_1 : Draft

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-2,818	2,893	,145	8,37	-102,656	398
Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
-17,114	-,139	16,975	-1121,683	6484,239	0

X_1 : Depth

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-49,586	2,65	,133	7,022	-5,344	398
Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
-55,468	-46,9	8,568	-19735,285	981384,207	0

Data for the period 880512. Unreasonable values and values above 0.2 m not included.

X_1 : Draft

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-2,876	2,078	,127	4,319	-72,267	269
Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
-9,972	-,076	9,896	-773,619	3382,47	0

X_1 : Depth

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-46,595	,885	,054	,783	-1,899	269
Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
-48,172	-45,38	2,792	-12534,148	584242,719	0

Data for the period 880609. Unreasonable values and values above 0.2 m not included.

X_1 : **Draft**

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-3,395	2,428	,284	5,895	-71,506	73
Minimum:	**Maximum:**	**Range:**	**Sum:**	**Sum of Sqr.:**	**# Missing:**
-11,366	-,09	11,276	-247,866	1266,039	0

X_1 : **Depth**

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
-48,235	,428	,05	,183	-,887	73
Minimum:	**Maximum:**	**Range:**	**Sum:**	**Sum of Sqr.:**	**# Missing:**
-49,132	-47,629	1,503	-3521,123	169853,032	0

Data for the period 880610. Unreasonable values and values above 0.2 m not included.

Produsert av GCS A/S, OSLO