

An experimental system for computer aided bird call recognition

VOLUME 2 Appendices A-B

Illan Samson Colombick

A dissertation submitted to the Faculty of Engineering, University of the Witwatersrand, Johannesburg, in fulfilment of the requirements for the degree of Master of Science in Engineering.

Johannesburg, 1992

Appendix A

The BCR software

Appendix A Table of Contents

A1 The BCR software	1
A1.1 Pipeline Processes	4
A1.1.1 SAMPLE	5
A1.1.2 SPECTRA	6
A1.1.3 SONAGRAM	11
A1.1.4 PEAKPICK	20
A1.1.5 BACKGND	22
A1.1.6 CLASSIFY	24
A1.1.7 ELEMENT	27
A1.1.8 SYNTAX	30
A1.1.8.1 CheckSyntax	31
A1.1.8.2 Elements: <i>ElementList</i>	34
A1.1.8.3 Candidate: <i>SyntaxSpec</i>	35
A1.1.8.4 The syntax tree: <i>SynTreeNode</i> and <i>SynTreeNodeList</i>	36
A1.1.8.5 <i>SyntaxSpec</i> .ElemSpecs: <i>ElemSpecList</i>	39
A1.1.8.6 <i>SyntaxSpec</i> .Init	41
A1.1.8.7 <i>SyntaxSpec</i> .CheckElements and <i>SyntaxSpec</i> .CheckSyntax	44
A1.1.8.8 <i>EntryPointList</i>	45
A1.1.8.9 Debug, Track and Trace features	48
A1.2 Support tools and utilities	57
A1.2.1 DRAWER and its support units ICON, MOUSE, PRTEGA	58
A1.2.1.1 MOUSE	58
A1.2.1.2 ICON	60
A1.2.1.3 PRTEGA	64
A1.2.1.4 DRAWER	67
A1.2.2 Bird call reconstruction: SOUNDGEN, LINKPEAK	77
A1.2.2.1 SOUNDGEN	77
A1.2.2.2 LINKPEAK	81
A1.2.3 The Bird Call Descriptor manipulator: BCDMANIP	91
A1.2.4 Libraries: GENLIB, OBJLIB, TEXTWIN	100
A1.2.4.1 GENLIB	100
A1.2.4.2 OBJLIB	107
A1.2.4.3 TEXTWIN	110
A1.2.5 Miscellaneous utilities: READELEM, READLPF, READSF, UPDATE, SONAPS	112

A1 The BCR software

The modules comprising the BCR system are described here. The descriptions are meant to highlight those aspects of the code that are not necessarily obvious on reading the source code listings. The listings (Appendix B) should be perused in conjunction with the corresponding material in this appendix.

Each module is described in a section on its own. If the module is the implementation of an algorithm described in the main document body, the appropriate chapter number is given in parentheses besides the name of the module in the headings. In those cases, the description here assumes that the reader has read the chapter in question. When, as with most non-pipeline-process modules, the software does not correspond to a section dealt with in the body, the description here is more general and complete.

The software consists of three types of program: BCR pipeline processes, BCR support tools and general utilities.

The **BCR pipeline** is a chain of processes through which the bird call data flows. Figure A1.1 illustrates the data flow. At the start of the pipeline is the **SAMPLE** process, whose input is the raw analogue bird call signal. Output from the **SAMPLE** process is the digitized signal, which is fed to the input of the next process in the pipeline, **SPECTRA**. **SPECTRA** performs a short-term FFT, producing the frequency-domain representation of the call. **PEAKPICK** extracts peak data from the **SPECTRA** output, **CLASSIFY** uses the peaks to produce the classified frame data, **ELEMENT** groups classified frames into a string of elements and finally **SYNTAX** tries to recognize the element string, producing names of species that match the call. The pipeline is thus the heart of the BCR system, and is all that is required to recognize a bird call.

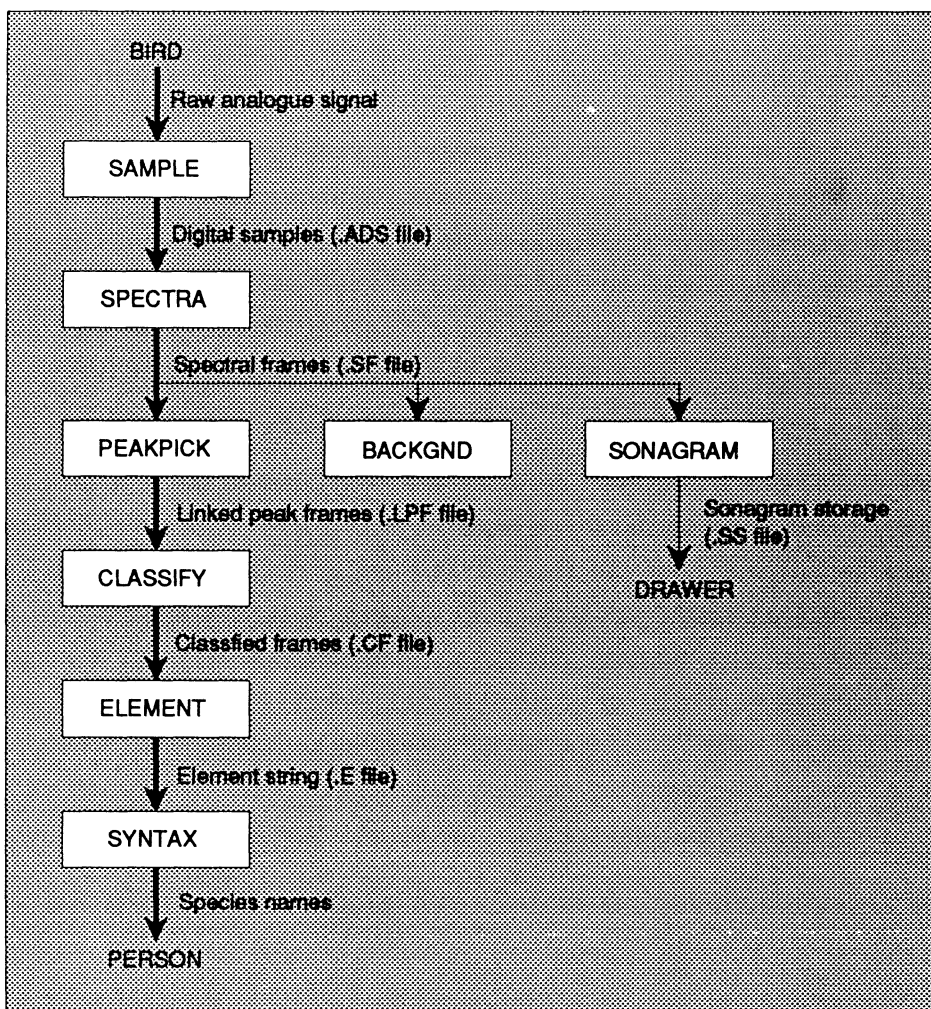


Figure A1.1 The BCR pipeline. Data flows in the direction of the arrows. Each process (named box) transforms the data in some way (name of data type is given next to arrows). SONAGRAM and BACKGND are special-case processes explained elsewhere in the text.

The **BCR support tools** are not a part of the recognition system, but are used to maintain an experimental environment in which to test that system. They may be software components such as SONAGRAM, which is like a pipeline process fed by the output of SPECTRA and producing a graphical representation of the bird call in the spectral domain. Or they may be a standalone program like DRAWER which is a graphical environment for the display and manipulation of the output of many of the pipeline stages, and of the SONAGRAM process. In all cases the support tools are closely coupled to the features of the BCR system, and have no use outside that context.

Finally **general utilities** are standalone programs or components which were written to augment the BCR system but are not specific to it.

The processes are all implemented as software components, Pascal **procedures**.

The procedures obtain input only from their parameter lists, and from the *Bird Call Descriptor*. The latter is a file recording the specific parameters of the "run" of a pipeline and allowing processes to communicate information other than the primary data to processes later in the pipeline. A process obtains data from the Bird Call Descriptor using the 'BCD' procedure contained in the BCDMANIP support tool (see section on BCDMANIP in this appendix). The BCDMANIP system allows carefully regulated and monitored access by processes to common data, as well as being a means for the experimenter to control and record runs.

A process performs its task by being called with parameters that specify where it is to find its "job" in the Bird Call Descriptor, as well as where its input and output are stored. It then uses 'BCD' to obtain any relevant information and processes the data according to this information and its algorithm. To run a pipeline, the process procedures are simply called one after the other in the correct sequence. To explain the functioning of a pipeline process, it is not necessary to know how the BCDMANIP system works, nor the details of where the bird data is stored. It is also important to note that the data files themselves contain no other information - such as "headers" - about what they contain, but consist entirely of a bird call which has processed in some way. Any non-call data reaches the process via its procedure parameters or the Bird Call Descriptor.

In the descriptions which follow, identifier names found in the listings are given in **bold type**.

A1.1 Pipeline Processes

As mentioned in the introduction to the appendix, the pipeline modules comprise the main part of the BCR software. Each process resides in a separate module. Each process takes its data from a file (usually left there by a run of the previous process) and its additional parameters from the Bird Call Descriptor. Each process is implemented as a Pascal procedure, accepting string parameters which determine the position in the Bird Call Descriptor where the process' additional parameters are to be found.

The string parameters work by specifying suffixes of file names. The file names are also the block names in the Bird Call Descriptor (see BCDMANIP for explanation of the Bird Call Descriptor and its use). Thus the procedure for the CLASSIFY process, **GetFrameClasses**, might be called as follows:

```
GetFrameClasses ('CALL1', 'A', 'B', 'C');
```

This would specify that the process uses the file names CALL1A.SF, CALL1B.LPF and CALL1C.CF. The .SF file name is used only to point to the appropriate additional parameters in the Bird Call Descriptor, placed there by the SPECTRA process when it created CALL1A.SF. The .LPF file is the input file of peak data, and the CALL1C.CF file will be the name of the output file containing classified frames. The .LPF and .CF names also refer to sections in the Bird Call Descriptor where the process may read or write parameter values. The convention, then, when running a process, is to specify names of relevant previously processed data files relevant to the current process, so that either the files or the parameters associated with them may be accessed.

The following sub-sections detail the pipeline processes. SONAGRAM is also described here - although not strictly a main pipeline process, it is of the same form. In general all processes are defined in units of the same name, and when referring to the listings or the functional descriptions, the name of the process and that of the unit are interchangeable¹.

¹ The concept of a unit is not standard Pascal, but may be thought of simply as a group of related Pascal TYPE, CONST, VAR, and Procedure and Function definitions. These definitions reside in a source file of their own and are compiled independently of other units or of any code which uses them. Use of a unit involves calls to routines (or access of variables) defined in the unit.

A1.1.1 SAMPLE

The SAMPLE process is the first process in the pipeline, and is thus responsible for the supply of digitized bird call data. Being the input interface to the world, it is hardware dependant, customised to the particular environment in which it is used.

Many different pieces of software were written or adapted for use with the BCR system as input sources and hardware changed. They are obsolete (they drive analogue-to-digital hardware which is unlikely to be used in the future) and have been provided with the system software but not documented. The current version of SAMPLE is also obsolete, converting from an old sampling system's format (ILS) to the BCR format. It's listing is given in Appendix B, as an example of the form of a future SAMPLE process. Rather than explain it's workings in detail, a specification of the required format and output of any future SAMPLE process is given here.

For maximum compatibility, the procedure which invokes the SAMPLE process should be called **GetSamples** and take an input (for the case of conversion of formats) and output file name as parameters. In order to work with the Bird Call Descriptor system, it should commence with a call to **PutCurrentProcess**, specifying the process name as 'SAMPLE'. It should record the sampling frequency in kHz in the 'Sampling_frequency:' field of the block named XXXXX.ADS in the Bird Cal Descriptor, where XXXXX.ADS is the name of the output samples file.

The SAMPLE process must output a file of sample values. The values are expected by SPECTRA (see next section) to be 16-bit signed integers, but limited to the (12-bit) range of -2048 to +2047. In fact it is probably possible to use the entire 16-bit signed integer range (-32768 to +32767) but that has not been verified. The range is understood by SPECTRA to be mapped (linearly) to the voltage range input to the A/D convertor. Thus if the instantaneous voltage range in the bird call signal is -5 to +5 V, then -5 V should result in the sample value -2048, and +5 V should give +2047. The output sample file should contain no headers or other data besides the 16-bit integers.

A1.1.2 SPECTRA

SPECTRA is the process which performs the FFT on the raw time-domain bird call data. Its main input is a sequence of sample values obtained from the analogue-to-digital conversion process being applied to a bird call (see SAMPLE). Its main output is a sequence of *frames* which together form the frequency-domain representation of the bird call (or rather, that part of the representation which is required by the BCR system). Each frame is the magnitude part of a discrete Fourier transform of a segment (also called a frame) of the input sequence. The input sequence is treated as a series of frames, each containing **WINDOWLENGTH** consecutive samples. Each input frame is transformed by padded FFT to an output frame of **FFTLLENGTH** values. **WINDOWLENGTH** and **FFTLLENGTH** are determined by the experimenter and the program as will be described shortly.

SPECTRA is invoked by calling the Pascal procedure **GetSpectra**, defined in the Turbo Pascal unit SPECTRA. In the following paragraphs the step by step actions of **GetSpectra** will be reviewed from a functional perspective.

The bird call sample sequence to be used as input is supplied to **GetSpectra** in the form of a disk file, and the output spectral frames are placed in a different file. This is true in general of all the processes - a single file contains the input at the time the process is invoked, and a new file is created containing the output when the process is complete. The names of these files are specified indirectly as parameters to the calling procedure. In the case of **GetSpectra**, the parameters are 'CallName' and 'SpecName'. **GetSpectra** builds file names of standard format from these parameters, and also uses them to find its "place" in the Bird Call Descriptor, where additional parameters for the process can be found. See the BCDMANIP support tool for the exact mechanisms. Thus, essentially, only these '..Name' parameters are supplied by the experimenter when invoking the process. Then the process has the responsibility of finding its main input and output files, and any additional parameters from the Bird Call Descriptor.

Inputs and outputs

The first action performed by **GetSpectra** is therefore to find its input, the file containing the digitized bird call. Before processing can start, however, there are a number of settings which have been placed in the Bird Call Descriptor by the experimenter, which control the segmentation of the call, and which must be checked and possibly adjusted. Any errors in the input specifications (such as incorrect file names, parameters out of range) will generate an error message and halt the SPECTRA process.

The first two of these additional parameters are **Start** and **Finish** (obtained from 'Start_time' and 'Finish_time' in Bird Call Descriptor) which specify (in milliseconds) the portion of the call that should be processed. **Start** = 0 and **Finish** = -1, for instance, will cause the entire call to be processed, while **Start** = 100 and **Finish** = 300 will process only the 200 ms section of sound produced 100 ms after the start of recording. This feature is useful for isolating only that section of the call which contains the bird call. If it were not provided, one would have to ensure that recording started and stopped at exactly the desired points. **GetSpectra** checks that these parameters are within a meaningful range of values and calculates two values to be used by the program from then on, **Starttime** and **Finishtime** based on them.

The remaining five parameters to **GetSpectra** are passed to another procedure, **SetWinFft**, for checking and adjustment. They are **WinDur** and its associated unit specifier **Wunits**, **FreqRes** and its unit specifier **Funits** and **TimeWindow**. They are derived from Bird Call Descriptor fields 'Window_duration:', 'Window_length:', 'Frequency_resolution:', 'FFT_length:' and 'Window_type:'. These parameters together determine the tradeoff between time and frequency resolution in output of **GetSpectra**. See section 3.2.2, The Short Term Fourier Transform, for more on this tradeoff.

Note that there is an extra interface layer, Bird Call Descriptor parameters being converted to intermediate variables which are then used to find values for global variables (or passed to **SetWinFFT** for checking). This is due to **SPECTRA** being adapted from previously written software, and other modules are far simpler. As far as the operator is concerned, only the Bird Call Descriptors need be understood. 'Window_duration:' is the frame length specified in milliseconds. If non-zero, its value is placed in **WinDur** and **Wunit** is set to **milliseconds**. If zero, the frame length is determined by 'Window_length:' and **Wunits** set to **samples**. A similar arrangement occurs with 'Frequency_resolution:' - it determines the number of points in the FFT, specified as a kiloHertz frequency value. If non-zero, its value is placed in **FreqRes** and **Funits** are set to **kiloherz**. Otherwise the actual length in points is taken from 'FFT_length:', placed in **FreqRes** and **Funits** set to **points**.

WinDur tells **GetSpectra** the size of the frames into which the input sequence will split. If **Wunits** is set to **samples**, the frames will be **WinDur** samples long. The variable **WINDOWLENGTH** will be set to this number. If **Wunits** is **milliseconds**, the routine calculates how many samples are needed to form an interval of this duration, and stores the answer in **WINDOWLENGTH**. Either way, **WINDOWLENGTH** then contains the number of points that will be read in each time from the input call and transformed to an output frame. The duration of the input frame is what determines the *time resolution* of the frequency-domain representation. As an example, if **WinDur** = 100 and **Wunits** = **milliseconds** and the sampling frequency used to record the input was 50 samples/second, **WINDOWLENGTH** will be $(100 \times 10^{-3}) \times (50) = 5$ samples.

FreqRes is the equivalent parameter to **WinDur** for specifying the length of the output frames. It determines the *frequency resolution* of the output. If **Funits** is **points**, **FreqRes** is the number of points in the output frame. That number is stored in **FFTLLENGTH**. If **Funits** is **kiloherz**, it is used to calculate the number of points (again stored in **FFTLLENGTH**) that are needed to effect a frequency resolution of **FreqRes** kHz. If, for example, **FreqRes** = 0.1 and **Funits** = **kiloherz**, and the sampling frequency was 500 samples/second then **FFTLLENGTH** should be $0.5/0.1 = 5$ points, because an FFT produces components ranging from 0 Hz to the sampling frequency, and thus if each component is to be separated (resolved from) its neighbour by 0.1 kHz, 5 components are needed to cover the range 0 to 500Hz. In fact, however, a value of 5 is not acceptable for a radix-2 decimation-in-time FFT, so **FFTLLENGTH** value will be rounded up to the nearest power of 2, i.e. 8, so that the eventual frequency resolution is better than 0.1 kHz.

Once all the range checking and adjustment occurs, the values of **FFTLLENGTH** and **WINDOWLENGTH** are used throughout the program. They are written back to the 'FFT_length:' and 'Window_length:' fields of the Bird Call Descriptor, so that the operator (and later processes) will have access to their actual (rather than specified) values. For convenience the derived values 'Frequency_resolution:' and 'Window_duration:' are also written back.

TimeWindow (obtained from the 'Window_type:' field) specifies what type of window function is to be applied to each time frame (i.e. to the segments of length **WINDOWLENGTH** points that have been extracted from the input). The choices are: **Rectangular** (frame is left as is), **Triangular** and **Hamming**. The Hamming window provides sufficient side-lobe suppression for the purposes of the BCR system (the triangular and rectangular windows are provided for experimentation purposes).

GetSpectra gets the value of the sampling frequency it needs for the above calculations from the Bird Call Descriptor, where it was placed by **SAMPLE** (see section on **SAMPLE** module). **GetSpectra** in turn places the values of **WINDOWLENGTH** and **FFTLLENGTH** it has calculated, as well as the implied resolutions (in ms and kHz respectively) in the Bird Call Descriptor.

Actions

Having obtained valid inputs, **GetSpectra** now begins the processing of the data. The action performed is summarised in the simple loop which forms the core of the routine:

```
Repeat
    writexy (70,1, Strung(0,0, Frame));
    GetNextTimeFrame;
    ApplyWindow;
    PerformFFT;
    blockwrite(Spectral_frames, SpectralFrame,1);
```

```

    GetMaxMag (newmaxmag);
    if newmaxmag > maxmag then maxmag := newmaxmag;
Until not AnySamplesLeft;

```

Each repetition of the loop processes one frame - that is, one time frame from the input file is transformed by FFT to an output spectral frame. The first line in the loop simply shows the number of the current frame. Then **GetNextTimeFrame** extracts the next **WINDOWLENGTH** samples from the input - storing them in an intermediate array. It maintains the count of frames processed, and will set the **AnySamplesLeft** flag to signal that all the input has been processed and the loop may end. **ApplyWindow** multiplies the points in the newly-acquired time frame by the window coefficients, as defined by a formula in function **Window**.

The next procedure, **PerformFFT**, calls the Turbo Numerical Methods Library routine **FFT** (renamed here to **FFT_TN**) to transform the time frame. Before doing so, however, the frame must be *padded*. Padding involves adding points to the time frame, owing to the fact that **WINDOWLENGTH** is not necessarily equal to **FFTLLENGTH**. The two lengths, it will be remembered, embody the time and frequency resolution of the system and are independent of one another. That independence is brought about by the mechanism of padding - if **WINDOWLENGTH** is 80, padding can add 38 0-valued samples to the frame to reach the nearest acceptable **FFTLLENGTH** of 128. Or the frame can be padded to 256, 512 etc if desired. At first it seems that this independence contradicts the general principle stated earlier, that the time and frequency resolutions must be traded off with each other in real systems. Indeed, increasing **FFTLLENGTH** results in more points per kHz in spectrum. But we are merely obtaining a better approximation of the Continuous Fourier Transform *of the truncated input signal*. It is the truncation - i.e. the fact that the Fourier transform of the window function does not have an infinitesimally narrow lobe - that determines the true limitation on the resolution of components in the signal.

The output of the **FFT_TN** procedure is the spectrum of the frame, from which the phase and negative-frequency portions have been discarded. The phase is considered unimportant for recognition purposes, because the aural systems of birds (and humans) is largely insensitive to phase differences. The negative frequency components are a mirror-image of the positive-frequency components because the time signal is real (not complex), and thus do not contain any additional information.

The **FFT** algorithm itself was modified marginally to scale the output, since there are different definitions as to the "correct" multiplication factor for the spectrum. In the present case, the output of the original formula (as defined at the start of section 3.2.3 , The Fast Fourier Transform) is divided by the square root of N. However this value is not of particular importance, since it is the relationship of components to each other that is required. To this end the last action in the loop is

performed (after outputting the calculated spectral frame), which is to find the component with the largest magnitude in the current frame. The magnitude is compared to the largest one found in previous frames, resulting, when the loop ends, in a record of the largest magnitude in the bird call. This value is recorded in the Bird Call Descriptor and serves as a reference magnitude for subsequent processes in the pipeline.

The final action performed by SPECTRA is to create a Single Component (.SC) file which will be associated with the spectral representation. The file contains the spectrum of a sine wave time frame. Thus it contains a single component, the shape of which is given by the spectrum of the windowing function. The component is positioned in the centre of the spectral frame, by using a sinusoid with frequency equal to half the sampling frequency. The purpose of the Single Component is to define the shape of an "ideal" component for a later process, PEAKPICK. This shape can be specified analytically, but by using **PerformFFT** to calculate it numerically, the calculation is made independent of changes to the SPECTRA process that affect the spectrum shape, such as use of a different window function.

A1.1.3 SONAGRAM

SONAGRAM uses the output of SPECTRA to create a graphical representation of the frequencies present in the call at each moment in time. The form of this representation is based on the graph drawn by a sonagraph. The spectral frames output by SPECTRA are a description of the frequency content of the bird call as that content varies over time (bearing in mind the trade-offs inherent in such a description, as explained in the first two parts of Chapter 3). This is exactly the information represented by the sonagram. SPECTRA is really a digital sonagraph, except that it produces output in a form which is meant to be manipulated by other computer programs, rather than understood by a human observer. SONAGRAM provides the latter facility, by converting the spectral frames into a graphical format. DRAWER is used to show this graphical representation on screen or paper.

The SONAGRAM input is the Spectral Frames (.SF) file that is output by SPECTRA. Its output is a file of values, Sonagram Storage (.SS file), which can be used to quickly draw a sonagram on screen.

The sonagram, as described in Appendix E, is a continuous trace. A point on the trace has a density (darkness) which represents the magnitude of a component, at the time and frequency indicated by the position of the point. SPECTRA output, on the other hand, is a number of discrete component magnitude values. They are discrete on both the time and frequency axes. That is, each frame represents a finite, discrete time range (given by the time resolution), and each point in a frame represents a finite, discrete range of frequencies (the frequency resolution). The primary function of SONAGRAM is to interpolate between these discrete points to produce a "continuous" output¹. Because of this interpolation, the Spectral Frames can be made into a sonagram of any convenient time or frequency scale - one can "stretch" or "zoom into" the data.

As well as performing interpolation calculations, SONAGRAM quantizes the magnitudes to sixteen levels, and groups adjacent components of the same magnitude to speed the subsequent rendering of the picture on the screen or printer. The following is an outline of the algorithm performed by **GetSonagram**, the procedure which implements the SONAGRAM process:

Inputs

A number of settings are read from the Bird Call Descriptor. The sampling frequency, FFT length, window length and maximum magnitude in the call, as determined by the previous SPECTRA process. The frequency scale (**FreqScale**), time scale (**TimeScale**), and maximum and minimum magnitude factors (**MaxFactor** and **MinFactor**) are values which have been placed in the Bird

¹ Since it is produced on digital equipment, the output can not be truly continuous. What is meant here is that it gives the impression of a continuous trace, taking into account the resolution of the display hardware

Call Descriptor by the experimenter, and are used to specify the scale at which the sonagram is to be drawn. SONAGRAM has two constants which tell it how to relate the **FreqScale** and **TimeScale** parameters to dimensions on the screen: **FreqBase** is the number of screen points (pixels) to use for a frequency interval of **FreqScale** kHz. **TimeBase** is the number of pixels to use for a time interval of **TimeScale** milliseconds. The values of **FreqBase** and **TimeBase** are fixed at sizes convenient for the average graphical display resolution, 600 and 200 respectively. A larger **FreqScale** then means that "more kHz" will be shown in the space of 600 pixels, in other words the frequency scale of the graph has been increased.

MaxFactor and **MinFactor** define a range, in fractions of the maximum signal magnitude, for the viewing of component magnitudes. If they are 0.2 and 0.7 respectively, then all components below 20% of the maximum magnitude will be shown as if they have zero magnitude, all components above 70% will be shown as having maximum magnitude (15), and all magnitudes in between will be shown with a magnitude somewhere in the range 0 to 15. By changing the range, the experimenter can see any desired detail in the variation of magnitudes, at the cost of no detail in the other parts of the total range. The quantization of all magnitudes to one of 16 levels is done because the magnitude levels are to be represented either as colours or brightness levels on screen, or as shades of grey printed on paper. Although some graphics screens handle more than 16 colours, printers usually cannot accurately represent more than 16 shades of grey, while retaining enough resolution to make the small shading variations apparent to the eye. The sonagram itself would not benefit significantly from a greater range of shading - analogue sonagrams typically are extremely limited in their ability to represent different densities.

Before processing the data, **GetSonagram** calculates scaling factors it requires for interpolation, based on the input parameters discussed above. Two factors, **PixelsPerKHz** and **LinesPerMsec** are written to the Bird Call Descriptor so that DRAWER will be able to label axes correctly without having to read any other values or do any recalculation. Two other factors, **TimeStretchRatio** and **FreqStretchRatio** are explained below in the context of their usage.

Frequency axis scaling and interpolation

Scaling the plot on the frequency axis involves scaling each spectral frame individually. The frames output from SPECTRA each have **FFTLENGTH** points. The purpose of scaling and interpolation is to generate a new frame which will represent the same spectral information (the same shape of magnitude spectrum) but do so using a different number of points. The reason for this is twofold: Firstly, the experimenter has specified a scale at which the plot should be viewed, via **FreqScale**. Secondly, the number of pixels along the axis line of the display device (defined in SONAGRAM as the constant **ScreenLength**) generally differs from **FFTLENGTH**. Since each point is to occupy one pixel (with the colour of that pixel being determined by the point's magnitude value), it is necessary to insert or delete points from the spectral frame.

The insertion or deletion of points, while keeping all points evenly spaced along the curve, is done by interpolating linearly between the original points, then "sampling" the continuous curve that results. Fig A1.2 illustrates this procedure, performed by the library routine **GraphTransform**, defined in the GENLIB unit (see later in this appendix).

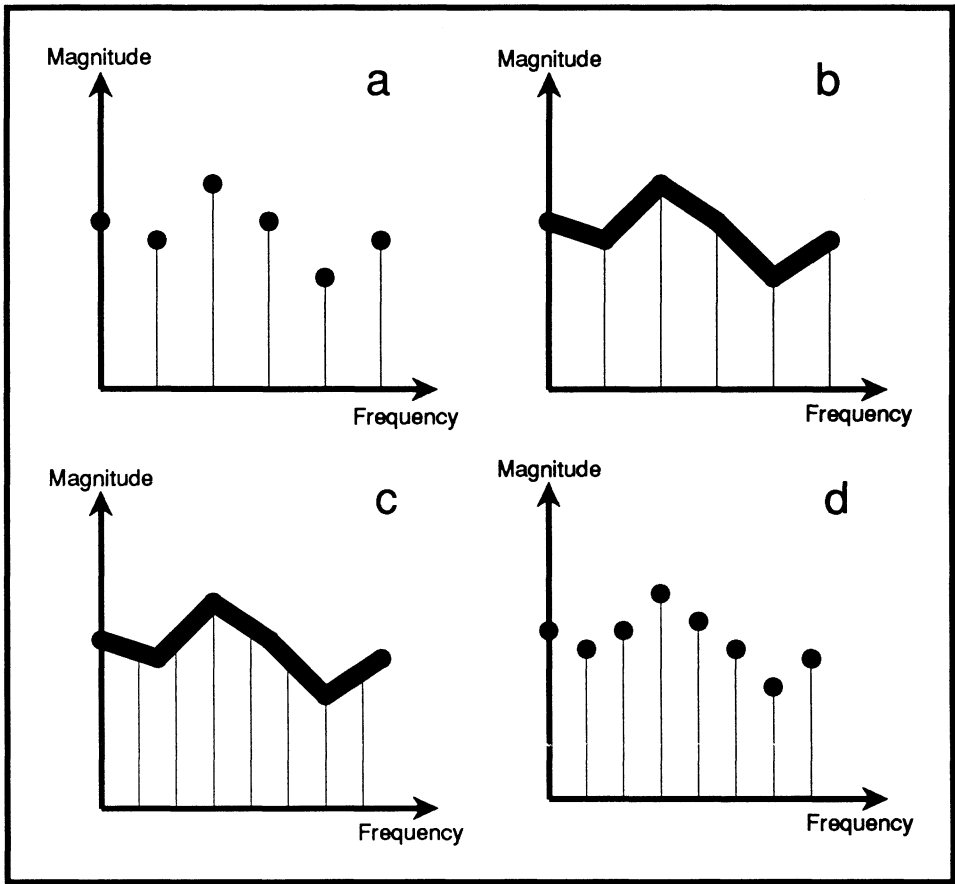


Figure A1.2 Interpolating a spectral frame. Frequency is on the horizontal axis, magnitude on the vertical axis. **a.** The discrete frame as it comes from SPECTRA. **b.** The interpolated frame. **c.** The new sample points are marked out. **d.** The new discrete spectral frame.

The parameters to **GraphTransform** determine exactly how many points the new frame will have. These values are calculated as in the following example, which assumes that the experimenter has specified a **FreqScale** value of 3 kHz, that **FreqBase** is 100 points, that the sampling frequency was 16 kHz, and that **FFTLLENGTH** is 256:

Since a 3 kHz range is to fit into 100 points, the desired point/kHz ratio is $100/3 = 33.33$. The spectral frame covers a range equal to the sampling frequency in **FFTLLENGTH** points, so the present point/kHz ratio is $256/16 = 16$. The number of points must therefore be increased by a factor of $33.33/16 = 2.08$. This is the **FreqStretchRatio** mentioned previously.

SONAGRAM now checks how many points arise when this factor is applied to **FFTLLENGTH/2** (only the positive-frequency half of the spectrum will be displayed, since the other half is symmetrical). If the resulting number of points is greater than **ScreenLength**, the full frame would not fit on the screen, so only the first part of the frame need be transformed. The number of points to take from the spectral frame is placed in **TransformFrom**. Conversely, if the whole frame fits in less than a full screen after scaling, the target number of points (in the new frame) is less than **ScreenLength** - this number is placed in **TransformTo**. **Graphtransform** is then called, being instructed to transform the first **TransformFrom** points in the original spectral frame to **TransformTo** points in a new frame array.

Displaying a single spectral frame

An interpolated spectral frame is displayed on the screen as if the points in Figure A1.2d were observed looking down the vertical axis. That is, the magnitude axis now projects perpendicularly out of the plane of the screen, rather than being aligned in the vertical direction. To depict distance out of the screen, the colour (or brightness) of the point is changed. Figure A1.3 illustrates the idea, also showing how the frequency axis is rotated from its horizontal alignment to a vertical one, as used in the conventional sonagram. DRAWER actually keeps the frequency axis horizontal for practical reasons, but that is not important in explaining SONAGRAM's actions.

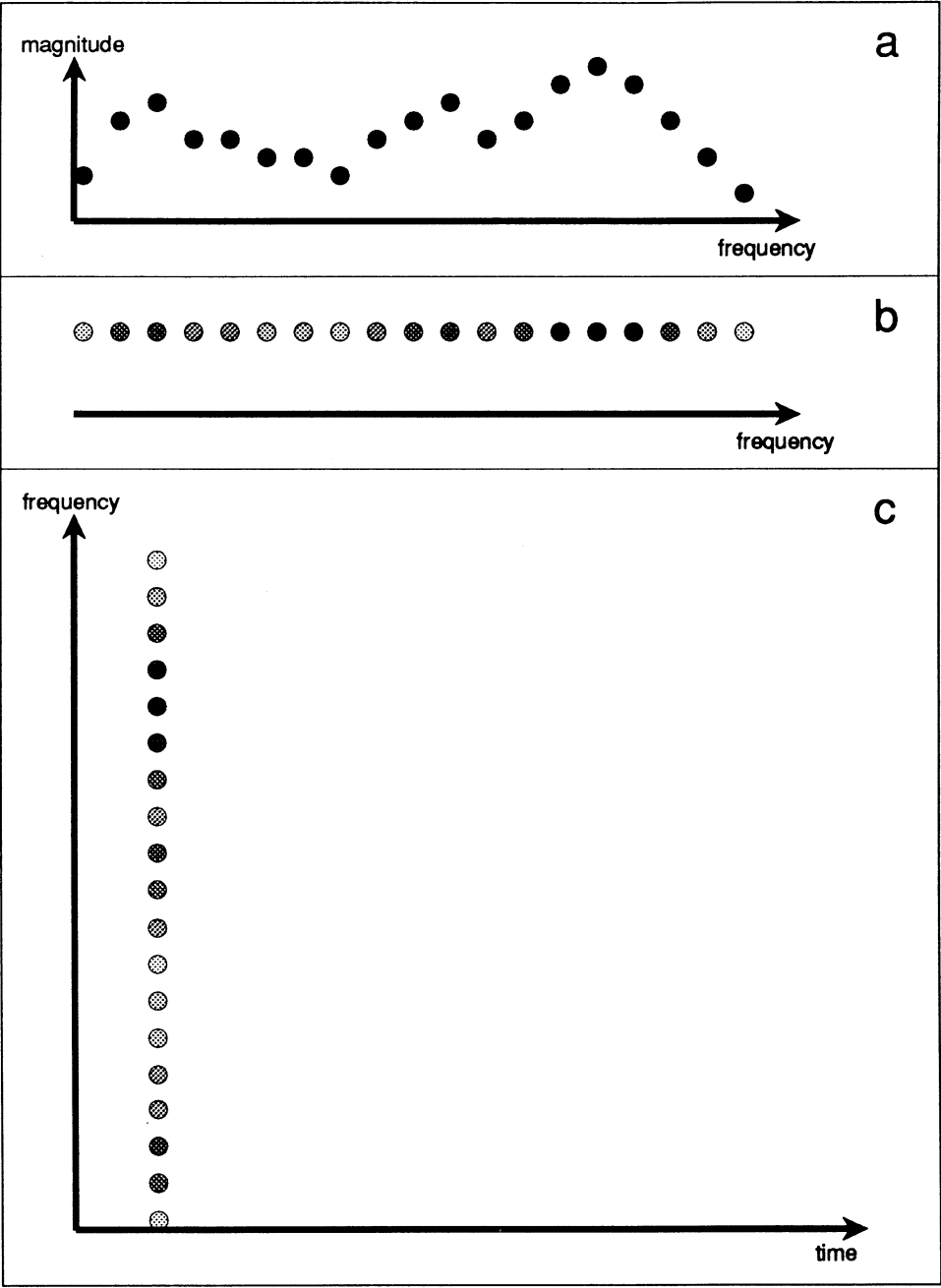


Figure A1.3 Using shading to represent the magnitude of points in a spectral frame **a**. Frequency on the horizontal axis, magnitude on the vertical. **b**. Frequency on the horizontal axis, magnitude is density of the point. **c**. Frequency on the vertical axis, magnitude is density. Horizontal axis can now be used for time.

If each successive frame is displayed as shown in Figure A1.3c, to the right of the previous frame, the spectral activity as it evolves over time appears. If a particular component is present for the interval of time covered by five frames, there will be five consecutive adjacent dark spots at the vertical height corresponding to the frequency of that component. These dots will join to form a short horizontal line. If the frequency of the component changes slightly between frames, the line will slant upwards, just as in musical notation successive notes in an increasing scale are placed in an upward slanting line. In this way a sonagram, suitably scaled on the frequency axis, is formed. Note that this mechanism is the complement of that used in the analogue sonagraph, where the picture is built from adjacent horizontal lines, each representing the time evolution of a single frequency range (see section 3.1.3, Real-world Spectral Analysis).

Time axis scaling and interpolation

The time axis must be interpolated for the same reasons as the frequency axis, and this is done using the same methods. However, since the points along the time axis are not all contained in one array, **GraphTransform** is not used. Instead the main part of SONAGRAM is loop which generates frames interpolated along the time axis, one at a time. The loop also converts these frames into a more readily rendered (and more compact) graphical form.

Before entering the loop, **TimeStretchRatio** is calculated. Each frame output by SPECTRA has information derived from a frame of **WindowLength** samples, so it corresponds to a time interval of **Windowlength/SamplingFreq** milliseconds. This accounts for one point along the time axis, but the experimenter requires **TimeBase** points every **TimeScale** ms. Thus:

$$\text{TimeStretchRatio} = [1/(\text{Windowlength}/\text{SamplingFreq})]/[\text{TimeBase}/\text{TimeScale}]$$

which tells SONAGRAM how many more or less new frames (vertical sonagram lines) are required than the number originally supplied by SPECTRA. To get these new frames, the algorithm interpolates just as **GraphTransform** interpolated to get new points.

To see how the interpolation works, consider a single position - say the 10th point - in a particular spectral frame. The 10th point in a new, interpolated frame is obtained by considering only the 10th point in the original (uninterpolated) frame that occurs before the current frame, and the one after. Figure A1.4 illustrates.

The magnitude **m** of a linearly interpolated point in a new frame is obtained by the intersection of the vertical line at the new frame position with the straight line drawn between point's magnitudes in the two surrounding old frames. Referring to Figure A1.4, the magnitude can be calculated (from similar triangles) as:

$$m_n[j] = M_n[i] + (d_j/D)(M_n[i+1] - M_n[i])$$

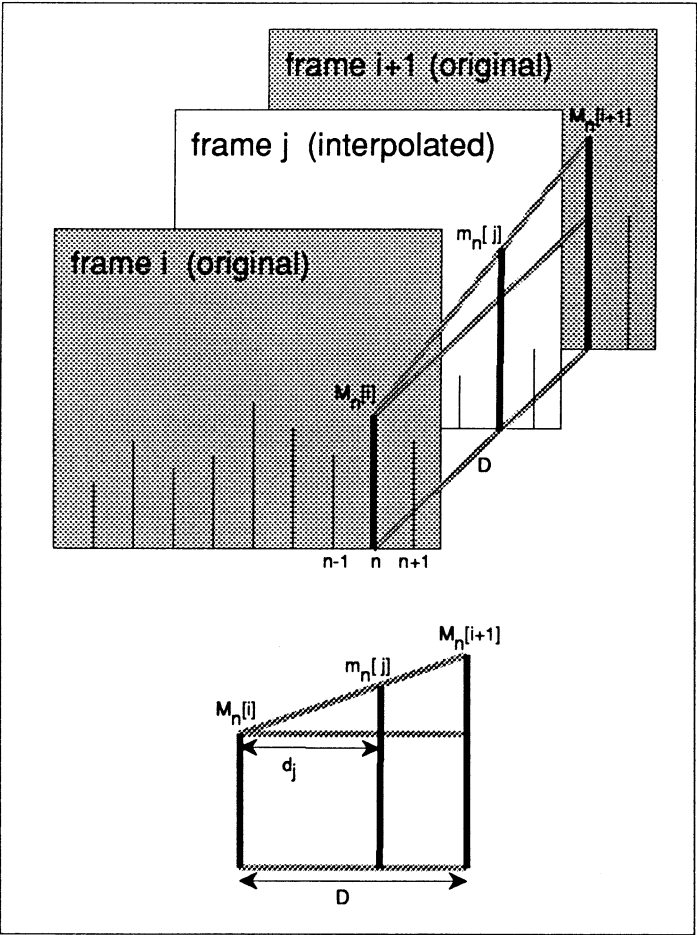


Figure A1.4 Interpolating between magnitude values of the n th point in successive frames. A thick vertical line (labelled $M_n[i]$) shows the magnitude of the n th point in the i th original frame. $m_n[j]$ is the magnitude of the n th point in the j th interpolated frame. Note that the original values themselves may never be "picked" to appear in the sequence of interpolated values. The thick grey lines are construction lines showing the interpolated value is calculated from the originals. d_j is the distance (along the time axis) from the j th interpolated frame to the previous original one, and D is the distance between original frames, i.e. the time represented by **WINDOWLENGTH** samples.

This equation is implemented in the SONAGRAM loop in the following way: **LeftPointWiseLine** points to the original (frequency-stretched) frame $M[i]$ and **RightPointWiseLine** points to $M[i+1]$. For each loop iteration the ratio d_j/D is calculated (in the variable **BetweenPt**) by adding $1/\text{TimeStretchRatio}$ to the previous ratio value. If the result is greater than one ($d_j > D$), this means that the new frame is to be calculated at a later point in time than the current rightmost original frame, $M[i+1]$. In that case, a new original frame is fetched from the spectral frame file (and

stretched along the frequency axis), so that **i** is effectively incremented, and 1 is subtracted from **BetweenPt** to reflect its new relation to the current original frame pair. The value of $M_n[i+1]-M_n[i]$ is calculated for all **n** and placed in the **DifferencePoints** array.

Whether or not a new original frame was fetched, the interpolated frame is obtained in **BetweenPointWiseLine** by letting **n** run from 0 to the number of points in the spectral frame, and calculating

$$\text{BetweenPointWiseLine}[n] = \text{LeftPointWiseLine}[n] + (\text{BetweenPt}[n])(\text{DifferencePoints}[n]).$$

After obtaining the interpolated frame, its values, which represent the magnitudes of points in the spectrum, are used to calculate **LineWiseLine**, whose values represent the colours (or shades) of points constituting one sonagram line. In effect, each point value is translated into a colour value, but two means are employed to reduce the drawing time and storage requirements of a line composed of such points. First, only one of 16 colours are used. Second, consecutive points of the same colour are merged and the length of the line segment needed to cover those points is stored along with its colour. These features are elaborated below.

Since display devices work with discrete colour values, the magnitude values need to be quantized. This is done before storing them, to minimise the computation necessary in the drawing program, and to reduce the storage space occupied by the values. The 16-shade range is sufficient to provide a smooth transition between colour values as explained previously. As also mentioned, SONA-GRAM allows the quantization to be limited to a subrange of the full magnitude scale, in order to display smaller variations in magnitude. The control of this subrange is through the Bird Call Descriptor parameters **MaxFactor** and **MinFactor**, which specify the magnitude values which should translate to shade 15 and 0 respectively, as fractions of the maximum magnitude in the call.

Thus for every point in **BetweenPointWiseLine**, the following action occurs: A shade level is calculated by scaling the point value, checking whether it falls in the required subrange, and either clamping it to 0 or 15, or truncating it to an integer in between. Then the level is compared with that of the previous point. If it is identical, the accumulated length of the line segment to be drawn is incremented. If the new level is different to the old, the current segment (a colour value and a length value) is left for later file storage as a **LineWiseLine** element, and a new segment is initialized to the new colour and length 1.

When the above loop terminates, **LineWiseLine**, representing a collection of different coloured segments of different lengths (in units of device pixels) to be drawn end-to-end, is written to the **Sonagram_storage** file. The colour level of the last segment has 16 added to it, so that it is the only one greater than 16. This will indicate to the drawing program that it is the last segment, and

therefore that the next segment belongs to a new line. In this way the file can contain lines with an arbitrary number of segments, although for practical purposes an upper limit is placed on the number by SONAGRAM. The use of line segments as opposed to points is efficient in storage when the average line segment is of length greater than 2 points. It allows for faster drawing of the lines whenever there are segments greater than 1 point in length, and the improvement is dramatic when long segments are present. The latter situation is almost always true of bird call sonagrams, since the spectral activity is often concentrated in one part of the frequency range, while the rest can be represented by long, 0-level segments.

A1.1.4 PEAKPICK

Inputs

The task of finding the significant peak positions in the spectral bird call data is given to the PEAKPICK process. PEAKPICK follows SPECTRA in the pipeline sequence. It accepts the spectral frames output by the latter, and in turn outputs *peak frames* - each peak frame contains the peaks found in one spectral frame. Additional parameters which PEAKPICK uses as input to control its algorithm are supplied via the Bird Call Descriptor. The **Background_Mag** parameter is the BCR system's estimate of the average background noise level. The estimate is performed by BACKGND, at the request of the experimenter using DRAWER as described later. By the time PEAKPICK is initiated, the estimate is assumed to have been made and a valid value placed in the Bird Call Descriptor. **SignalToNoise** is the SNR which is assumed to apply to the call. This value is placed in the Bird Call Descriptor by the experimenter. Other parameters of the call (number of frames, sampling frequency etc) specified at earlier stages are also collected by PEAKPICK before processing begins.

One final source of external input is the Single Component - the spectrum, as output by SPECTRA, of a signal consisting of a single sinusoid. SPECTRA places this reference spectrum in a file whose name PEAKPICK is able to derive from the bird index information passed when it is initiated - via the **PickPeaks** procedure call. The Single Component data gives the spectral shape of the lobe produced by a single tone in the call. It is this shape that is subtracted from the call's spectral frame when PEAKPICK has decided on the position of a peak in that frame. The frequency of the sinusoid used is half the sampling frequency, so that the peak is always centred in Single Component data sequence (i.e. the point with the largest magnitude has an index of $\text{FFTLLENGTH}/2$). The magnitude of this largest point is fixed at 1 since it will be scaled before use.

Actions

The action performed by the peak extraction procedure, **PickPeaks**, is extremely simple: A spectral frame is read in, and scanned to find the point with the largest magnitude. That point's position and magnitude are passed to the **SubtractPeak** procedure which removes a lobe of the correct scale at that position. The frame is again scanned for the largest magnitude, **SubtractPeak** is again invoked, and the process repeats until altogether **PeaksPerFrame** iterations have occurred. **PeaksPerFrame** is not strictly an input, its value being fixed at 5 in the code, but could be moved to the Bird Call Descriptor if adaptive peak extraction or experimentation is required. Each maximum found is recorded in an array, **peaks**, of type **peakframe**. A **peakframe** is defined in the common types unit BCRTYPES and records not only the position and magnitude of up to **PeaksPerFrame** peaks, but also an index which links them to peaks in the previous frame. This

index is placed by LINKPEAK and used by SOUNDGEN, as explained later. It is initialised to zero by PEAKPICK. Before the next frame is read in, **peaks** is sorted by frequency and written to the Peak Frames output file. The sorting is handled by procedure **SortFreqs** and is a bubble sort.

When scanning the frame for the largest magnitude, only those points which have magnitudes exceeding the threshold of significance are considered. The threshold is calculated as the product of the observed average background noise and the assumed signal-to-noise ratio, as given by the input parameters mentioned above. Thus only points that are higher than the background level by a factor equal to or greater than the SNR are candidates for peak points. If there are less than **PeaksPerFrame** such points (remembering that many points are removed in between scans, by **SubtractPeak**) then the frame will simply be considered to contain less than the maximum number of peaks. The sorting of **peaks** places all non-zero peaks at the start of the array, so that they can easily be counted by a process which needs to do so.

SubtractPeak works by subtracting, from the magnitude of each point in the **SpectralFrame** array, the magnitude of a point in the **ComponentSpectrum** array. The latter contains the spectrum of the sinusoid created by SPECTRA (single Component) as described previously. The parameter passed to **SubtractPeak** is the position in the spectral frame of the maximum (peak) which is to be eliminated. This point's magnitude is used to scale all **ComponentSpectrum** magnitudes, so that the maximum magnitude of **ComponentSpectrum** (which is 1 before scaling) coincides with the eliminated peak magnitude. The **ComponentSpectrum** is shifted so that it's point with the maximum magnitude (located in the middle of **ComponentSpectrum** before shifting) coincides with the eliminated peak's position. Because the spectrum stored in **ComponentSpectrum** is shifted before use, it must have "space" for more than the $\text{FFTLLENGTH}/2$ points strictly necessary. It is thus left as generated by SPECTRA, with the full **FFTLLENGTH** points. The shifting itself is done by modifying the index to **ComponentSpectrum** rather than physically moving the data, but the larger array is still necessary.

A1.1.5 BACKGND

BACKGND is not a pipeline process, in that its output is not a processed form of the bird call, but a single parameter in the Bird Call Descriptor. Nevertheless it is an essential part of the pipeline, and must be invoked at least once for every bird call, so that it is included in this section together with the pipeline processes. The parameter output by BACKGND is 'Background_Mag:', the estimated average magnitude level excluding the contributions of the bird call sounds themselves. In its current form, BACKGND is invoked by the experimenter interactively, from the DRAWER program. The experimenter views the sonagram of the bird call, and (graphically) selects a portion of the signal which seems to be representative of a silent (call-less) interval. The choice of interval is subject to the judgement of the experimenter and is therefore not objectively defined. It is nevertheless a rather easy and clear choice to make, which is in contrast to the difficulty of automating this choice in software. For now, the human recognition capability is being used to simplify the work of the BCR system.

The problem in automatic background detection centres around the definition of significant sounds. While the experimenter knows with fairly great certainty what sounds are intended as the target for recognition, there can be no simple criterion for use in an algorithm. In many cases, the target sound is louder than all others, but not by a predetermined amount. In some cases the target bird is alone in calling, but in some there are secondary bird calls which interfere with the main signal. It is a simple matter for a person to indicate an interval during which the disturbing signals are present without the main signal (even in the field this should be a straightforward matter), but a complex one for a machine. Once this interval is indicated, 'Background_Mag:' can be calculated and used as a guide to differentiation of disturbing signals from the target signal in the remaining portions of call.

Inputs

The main input to BACKGND is the spectral frame data output from SPECTRA. The activating procedure for the process, **BackGround**, takes two parameters that indicate the start and end of the interval over which the average magnitude is to be found. The parameters are specified in terms of frame number.

Actions

Each frame is read in, the magnitudes of all its points summed and the sum is divided by the frame length (half the FFT length, as read in from the Bird Call Descriptor) to give an average magnitude for the frame. This average is added to an accumulated sum, which is divided by the number of frames in the interval after all the frames have been read. The result is the average magnitude for the interval, and this is written to the Bird Call Descriptor.

Before establishing that it is the current process (which allows the BCDMANIP system to control access to Bird Call Descriptor fields) it saves the value of the current process that prevailed when BACKGND was invoked. This value is restored on BACKGND completion. This is done to allow BACKGND to be invoked from within another process, as indeed is done (from inside DRAWER). Other processes (of the pipeline and utility type) do not need to do this. See the BCDMANIP system described later for further insight into the current process mechanism.

A1.1.6 CLASSIFY

Inputs

The three main input parameters to the CLASSIFY process are threshold values obtained from the Bird Call Descriptor. The first is 'Distance_thresh:' (stored in variable **DistanceThresh** in the code), which is the standard deviation threshold used in classifying single and harmonic frames, as described in section 5.1.1, Frame Classification Scheme. It is specified in kHz. The second and third parameters are thresholds provided to fit in with the philosophy of independence of processes. 'High_pass_freq:' is a kHz value. 'Class_mag_thresh:' is a magnitude value, specified relative to the average background magnitude ('Background_mag:' in the Bird Call Descriptor). Peaks with frequency lower than 'High_pass_freq:' will not be counted. Frames with a mean peak magnitude less than 'Class_mag_thresh:' will be classified silent. This allows the classification algorithm to pare off peaks which are likely to disturb its function, namely low amplitude and/or low frequency peaks which are not in the range of normal bird calls and are thus probably noise. PEAKPICK need not be concerned with which peaks are significant *for classification purposes*, and is free to concern itself with more general significance considerations (such as the SNR value). Should one of either the PEAKPICK or CLASSIFY algorithm need to be changed at a later stage, there should be few repercussions for the other. Other inputs are 'Background_mag:' (for use with 'Class_mag_thresh:') and 'Number_of_frames:' (for input file reading purposes).

Actions

The CLASSIFY process is invoked through a call to **GetFrameClasses**. The action of **GetFrameClasses** is to read in a peak frame at a time, remove unwanted peaks, count those left, calculate mean and standard deviation values and decide on the class of the frame. After the decision, further class-specific tests and calculations are performed to confirm or refute the classification. Then the classified frame, which consists of the class and a characteristic magnitude and frequency, is output to file before the cycle begins again with the next frame. The fact that the peaks in the incoming peak frame are sorted in ascending frequency order is relied upon by **GetFrameClasses** in counting the peaks and in checking on the inter-peak distance.

Counting peaks: Starting from the lowest frequency peaks, the number of peaks with frequency below the high pass threshold is counted. The number of usable peaks in the frame, **PeakCount**, is now obtained by subtracting this from the total number of peaks. Note that this method relies on the fact that the frequencies of all the 'empty' slots in the peak frame have been set to zero and the frame has been sorted in ascending frequency order. The peaks below **HighPassThresh** are removed by shifting the higher ones down so that the non-zero peaks in the frame occupy the first and subsequent slots, with the empty slots (assuming the frame has less than 5 peaks) bringing up the rear.

Calculating statistical parameters: General functions **Mean** and **StdDev** defined in GENLIB are used. The functions accept two arrays of real numbers (any size) - one for the data and one for weights, as well as parameters specifying how many points are in the arrays, and whether weighting should be used or not. For the first set of parameters, the mean and standard deviation of the peaks, the frequencies of the peaks are simply transferred to the data array (called **freqs**) and the magnitudes to the weights array (called **mags**, although weights need not necessarily be magnitudes). The results are stored in **freqmean** and **freqstdv**. The mean peak magnitude is obtained by passing the magnitudes as the data, a dummy array as the weights and turning weighting off. This value is stored in the **mag** field of **classification**, which is the classified frame variable that will be written. The **mag** field is the characteristic magnitude for the frame. The **classification.mag** value is checked to see if it below the class magnitude threshold - if so, **freqmean** is reset to 0, which is a signal to a later portion of the program that the frame should be classified silent.

For the second set of parameters, the mean and standard deviation of the inter-peak intervals, the differences between adjacent peaks are passed as the data in **freqs** to **Mean**. These differences are simply the differences between consecutive frequencies in peak frame variable, because the peaks have been arranged in ascending frequency order. The first "difference" is that between the first peak and an imaginary peak at 0 Hz. The weights are calculated as the geometric means of adjacent peaks, and passed in **mags** to **Mean**. The first weight is equal to the magnitude of the first peak. The results are stored in **diffmean** and **diffstdv**.

Preliminary classification: The scheme of Figure 5.3 is now implemented. The class is silent if **freqmean** is 0, as mentioned above. If not silent, it is single if **freqstdv** is below **DistanceThresh**. If not single, it is harmonic (**harmoni** in the code) if **diffstdv** is below **DistanceThresh**. And if not harmonic, it is one of the other multiple classes, preliminarily assigned the multiple (**multi**) class here. The characteristic frequency field (**classification.freq**) is preliminarily assigned to **diffmean**, although it will remain so only if the classification of 'harmonic' is retained in the final stage below.

Final classification: The relation between peaks in a frame that has been preliminarily classified as harmonic is double-checked: The first peak must have a frequency nearly equal to **diffmean**, i.e. the fundamental. Failing this, the second peak (perhaps the first peak is noise or spurious) should have this relation. In case the fundamental is missing in this particular sound (filtered out by the bird's vocal tract), the first and second peaks are also checked against the second harmonic frequency value. If none of these checks are positive, the classification reverts to multiple. This double-checking is a vague safeguard against a chance correlation between some higher peaks in a multiple frame being misinterpreted - it is probably not essential, but it might pick up an occasional classification error.

If, after the double-check, the frame is not harmonic, the characteristic frequency is set to **freqmean**, which applies to all other classes. If the class is **multi**, the number of peaks is now checked to determine whether it is in fact a noisy or a multiple frame. If **PeakCount** is greater than **PeaksPerFrame-1**, the classification is **noise**, otherwise it is left as **multi**. Note that for the minimum case of **PeaksPerFrame = 5**, this means that a non-harmonic frame is considered noisy if it has 4 or more peaks. Strictly, the number of peaks required to imply noisiness should probably be made independent of **PeaksPerFrame**, with another parameter to the peakpick process.

Output: The **classification** variable is written to file.

A1.1.7 ELEMENT

Inputs

The Bird Call Descriptor parameters 'Window_length:' and 'Sampling_freq:' are used to convert numbers of frames into durations in seconds. 'Number_of_frames:' is used to indicate processing progress. 'Dominance_ratio:' is the only specialised parameter for this process - it indicates the minimum proportion of frames of a particular class that an element must have to be labelled as belonging to that class. Thus if an element consists of 40 harmonic frames out of a total of 60 frames, i.e. is 66% harmonic, and 'Dominance_ratio:' is 50%, the element's class will be set to harmonic.

Actions

The ELEMENT process is activated by calling the **DetermineElements** procedure. It works by reading classified frames one at a time, using the information in the frame to accumulate data for the current element. The action of accumulating data about a frame will be referred to here as *counting*. When a frame is encountered that represents an element boundary, the current element is processed and then is written as output. The boundary frame is then counted as the first frame of a new current element.

The algorithm distinguishes between two types of element (and frame) - non-silent, and silent. A frame's **class** field is examined to determine its type. The variable **currentsilent** keeps track of the type of the current element - it is TRUE if the current element is silent, FALSE otherwise. Other variables which accumulate information about the current element are **ElemFreqSum** and **ElemMagSum**, to which a frame's characteristic frequency and magnitude are added each time it is counted, **ElemFrameCount** which tallies the number of frames in the current element, and **BreakDown**, an array that keeps track of the proportions that the different classes represent in the current element, information which will be used to form the element's class profile.

Silent elements have no class profile or average frequency - the only parameter of relevance to them is their duration. Thus to count a silent frame, it is necessary only to increment **ElemFrameCount**. To output a silent element, a variable of type **elementtype** has its **occurrence** (profile) and **freq** (frequency) fields set to 0, its **class** set to silent and its **durn** (duration) set to the number of milliseconds calculated from the value in **ElemFrameCount**, and is then written to file. These output actions are collected into a separate procedure, **WriteElement**, and are common to the output requirements of both silent and non-silent elements. Non-silent elements require extra processing before being ready for output, and these actions, explained later, are performed by **WriteNonSilent** which then calls **WriteElement** to do the actual output.

To count a non-silent element, a task performed by the procedure **CountNonSilent**, not only must **ElemFrameCount** be incremented, but the frame's **mag** field must be added to **ElemMagSum**, its **freq** field to **ElemFreqSum** and its class incorporated in **BreakDown**. The former two actions are self explanatory. Counting a frames class involves adding to one of the four components of the **Breakdown** array - one component for each class. Rather than simply incrementing the value, thus counting the occurrence of one of the classes, the value is increased by the frame's characteristic magnitude. This means that a frame counts more if it is louder, as explained previously in section 5.2.2, Element Characterization Algorithm. The class profile is thus weighted by the frame's magnitude. This is the reason for accumulating the **ElemMagSum** value; it is the total against which the sums in **Breakdown** will be compared to give a normalised profile - see **WriteNonSilent**.

The main loop thus works as follows: A new frame is read in. Its class is compared with that implied by the value of **currentsilent**. If the two are the same (i.e. **currentsilent** is FALSE and the new frame is silent, or **currentsilent** is TRUE and the new one is silent), then the new frame is simply counted. If it is silent, counting is simply incrementing **ElemFrameCount**, and if it is non-silent, **CountNonSilent** is called which updates all the other accumulator variables.

If, however, the two classes are different, the new frame is the start of a new element. Before counting it then, the current frame must be output. If the current frame is silent, output is just a call to **WriteElement**, passing "silent" as the class parameter and "0" as the frequency parameter. In other words, writing a silent, 0-frequency element whose duration and profile are obtained from the global **ElemFrameCount** and **BreakDown** - the latter is empty in this case. If the current frame is non-silent, **WriteNonSilent** is called.

WriteNonSilent divides the all the class totals in **Breakdown** by the **ElemMagSum**, normalising the class profile. Thus if the harmonic component of **BreakDown** was 4000, being a result of 40 frames each with magnitude 100, and the total magnitude sum is 5000, then after normalisation, the harmonic content is 0.8, indicating a weighted 80% harmonic content for the element. The largest component in **BreakDown** is now found, and the corresponding class stored in **maxclass**. Now **WriteElement** is called, with the class parameter set to **maxclass** if that class's proportion in the element is at least as great as the 'Dominance_ratio:' parameter. The frequency parameter is set to the average characteristic frequency in the element, i.e. $\text{ElemFreqSum}/\text{ElemFrameCount}$.

If the proportion of **maxclass** is less than that specified by 'Dominance_ratio:', in other words there is no clearly dominant class in the element, then the elements class (the value passed to **WriteElement**) is set to **mixed**. The mixed class was not encountered until now because it is one which pertains the spread of different types of frame rather than the type of frame itself. There is no mixed class for frames, only for elements.

As soon as a new element boundary is detected, **WriteElement** is called, either directly by the main loop, or through **WriteNonSilent**, to output the old element. **WriteElement** is therefore also an appropriate place to reset all the accumulator variables - all running totals are set to 0. Thus when the main loop counts the boundary frame, it is counting for a new current element. The only variable that is not reset is **currentsilent**, which is toggled - at every element boundary the current element must go from silent to non-silent or vice versa.

Before entering the main loop, the algorithm "peeks" at the first frame - reads it but does not consume it. **currentsilent** is set according to the class of that frame. This controls the behaviour of the loop after reading the first frame. If this were not done, the first frame would sometimes trigger an element output (if its class was different from the **currentsilent** setting) and sometimes not. Instead the "peek" ensures that no element write occurs immediately, the first write instead occurring on the first true element boundary. Thus the first element in the string output by **ELEMENT** is always one which starts at the beginning of the call and ends on the first element boundary. It is a partial element, and not of much use to the syntax matching stage, but since the first element is consistently partial, it can always be discarded. If no peeking was done, the partial element would sometimes be the second one, with the first being the extra, spurious one written at the very beginning. The last element in the output string is similarly always partial, and can also be confidently discarded by the syntax stage.

A1.1.8 SYNTAX

The SYNTAX process is the end of the BCR pipeline. It is not like a simple process which accepts one primary input and outputs a processed form of that input. Rather it consumes the input element string from the pipeline, as well as referring to a potentially large database of static bird call syntax information, and returns a set of scores that can be used to decide the species associated with the pipeline input. While from the pipeline's point of view it is simply the last process, from its own frame of reference it appears as a large, independent program dealing with a number of different types of input, outputting an "intelligent" decision and also displaying and recording its own workings for diagnostic purposes.

The program is the largest of the software entities in the BCR system. Besides the relative complexity of the algorithm it embodies (compared to the simple tasks performed by other pipeline processes) it has the task of translating the human-oriented text of the syntax specifications into an algorithmically efficient internal data representation. It must handle element strings and syntax trees of indefinite length, necessitating the construction of special data structures. Finally the software is required to report its complex workings to the user, so that its operation can be analysed and validated. The latter function is not necessary for the experimenter who works with a finished-product BCR system and can rely on its correct operation. However in the prototype stage, and while changes might well be made, it is vital to be able to track the algorithm's operation as closely as possible. The program undertakes to provide this feedback in a visual, interactive form to enhance its interpretation by the user.

Because of its greater complexity the program is well served by the use of object-oriented programming (OOP) in its design and construction. Most software modules in the BCR system were too small to require OOP design. The other large modules, SPECTRA and DRAWER, might have benefitted from the OOP approach, but was historically constructed before the principles of OOP were publicised with the advent of widely available object-oriented programming languages. The data access control mechanism of the BCR system itself, as implemented by BCDMANIP (described later in this appendix), is basically an object-oriented system. The Bird Call Descriptor is an object, which exists independently of the existence of the processes, and which deals out or accepts data to processes through a message-based interface, while transparently performing a number of error and type checking actions.

SYNTAX, being the most recently written software, was able to take advantage of the availability of native OOP languages, specifically Turbo Pascal 5.5 and above. The effect of such languages is not to ensure OOP design, but to greatly facilitate implementation when such design principles are used. In the SYNTAX program, three separate types of objects can be found: Instances of a

general class of list objects, instances of a general class of screen window objects to aid in the display of the workings of the algorithm, and a number of custom-designed objects to embody the bird call data and provide the functionality of the syntax checking system.

Because of the size of the software it is not desirable to describe the SYNTAX process in the same technical detail as other modules. The different flavour of OOP code also points towards a different descriptive approach. Fortunately, the increased encapsulation and strict interface specification of OOP allows such code to be described fully at a higher level. The details of implementation are less crucial to the understanding of the code, even when the reader must have enough knowledge to be able to implement changes and additions. Indeed, the reduced importance of understanding incident programmer's "tricks" and stylistic quirks greatly enhances the maintainability of the code, and the omission of the fine details in this description should in fact enhance the reader's understanding¹.

A1.1.8.1 CheckSyntax

The syntax checker is activated by calling the **CheckSyntax** procedure. On invocation, two file names are passed as parameters. These are used to open two file variables, **ElementFile** and **Candidates**. **ElementFile** is the element string obtained from previous pipeline process, **ELEMENT**. **Candidates** is a list of candidate syntax specs to be compared with the input element string. The names of the candidate syntax spec files are listed one after another, each on a separate line. The extension of a syntax spec file is assumed to be 'S' and is not included in the **Candidates** list. Thus if candfile contains the following text,

```
REDB
SOUT
MASK
```

then the syntax checker will compare the element string in **ElementFile** with the syntax specs in the text files named REDB.S, SOUT.S and MASK.S, obtaining a set of scores for each. The scores are output to the standard output device in one of three formats - a detailed track of the parse, a table of scores meant to be easily read by people, or a table formatted for easy reading by a computer program. The format choice is governed by a compiler symbol (i.e. a different version of SYNTAX must be compiled for each type). The track format is explained in section A1.1.8.9, the score formats are detailed later in this section.

¹ These advantages are, needless to say, some of the prime motivations for the use of OOD in programming.

An even more detailed diagnostic arrangement than track mentioned above is enabled when the compiler symbol **Trace** is defined. Then, an on-screen display of the progress of the parse is provided, and the user can step through each action of the algorithm. A second symbol, **Debug** is also available, although as its name implies it is purely a debugging device. It causes the various objects in the program to report their contents prior to destruction - a validation of the correct operation of the memory allocation and deallocation. The diagnostic features are implemented by scattered formatting and output instructions throughout the code. The extra statements, together with the compiler directives that link them to the appropriate symbols, are interruptions in the flow of the main algorithm, reducing the readability of the code. Therefore, a code listing has been provided in the listings appendix (Appendix B) from which all the diagnostic code has been removed. The result is the code that would be compiled if all the diagnostic symbols were turned off. This is the listing that will form the basis of the following explanations. The diagnostic features are explained in a general way in section A1.1.8.9, where the full listing, which is of course also provided in the appendix, can be used as reference.

The inputs to the syntax checker are, in summary, the element string file and the candidate list file, and through the latter the syntax spec files. The outputs are the scores or match result track on the standard output, and/or the detailed parse progress on screen. No Bird Call Descriptor fields are used.

CheckSyntax is the main (and only top-level) procedure in the module - all other actions are performed by objects' methods. The discussion therefore begins with a description of **CheckSyntax**'s actions, which will introduce the two main objects in the module, **Elements** and **Candidate**. The objects represent the two main players in the recognition process, the bird call element string, and a candidate syntax specification, respectively. For the purpose of understanding **CheckSyntax**, the overall form and function each object will be outlined. Then, the detail of each object will be exposed in the following sections.

The **CheckSyntax** procedure starts by initialising the object that will represent the element string, **Elements**, which is of type ***ElementList***¹. The initialisation of **Elements** is performed by its **Init** method which accepts the name of an element file (**elementfiletype** = file of **elementtype** = file of the format output by the ELEMENT process). **Init** reads the information in the file into the internal ***ElementList*** format (see later descriptions).

¹ Object type identifiers here will appear in bold italic face to distinguish them from object instances (variables) and from ordinary type identifiers (which appear in the usual bold face of all other identifiers).

The output result file (**output**, set to standard output) is next opened. What is sent to this file is determined by the compiler symbols **HumanResult** and **MachineResult**. The effect of **MachineResult** is to cause **CheckSyntax** to print three columns of numbers (preliminary score, failure score, match score) for each candidate, with no labels or formatting text. Candidate tables are separated by blank lines. This means that a program such as a spreadsheet, or any other software, can easily read the result file and use it to present the results in another way. When **HumanResult** is defined, which it is by default, the same information is printed but with additional formatting characters and information such as candidate bird names. This is meant to be understood more easily by humans. **HumanResult** is assumed to be defined in the discussion that follows.

The **output** is initialised with the headings of the score table it will contain - one line for each candidate, organised into several columns. The first column is the name of the candidate, followed by an entry indicating whether the essential elements were present in the string or not. Then the preliminary score, the failure count (headed "SYN FAILS" for "syntax failures") and the appropriateness score or "SYN SCORE" ("syntax score"). Next, the candidate list **Candidates** is opened. Any errors encountered in the opening of the files are trapped and handled by the **runerror/UserMessage** mechanism defined in the GENLIB unit. The mechanism halts execution with a run-time error 255, printing the desired message on the screen together with the location the error occurred. When using the Turbo Pascal integrated debugger, the position of the **runerror** line in the source code is located, greatly aiding interpretation of error messages.

The initialisations finished, **CheckSyntax** enters a loop, each iteration of which compares a different candidate syntax spec with the element string. The first action of an iteration is to re-initialise the **Candidate** object so that it contains the information from the next syntax spec in the list. The **Candidate** object is of type *SyntaxSpec*, which is the type designed to represent all the information in a syntax spec, as well as containing the methods required to perform the syntax checking algorithm. Its **Init** method accepts the name of syntax spec text file and reads it, checking for format errors and translating the text into an internal data representation. The name passed to **Init** on each iteration of the loop is obtained from **Candidates** (with the extension '.S' appended).

Once the **Candidate** is initialised, its **BirdName** field contains the name of the bird which appeared in the syntax spec. This name is therefore placed at the start of the **OutString** string, which will be output as a line of text to the **output** file at the end of the iteration. The use of the Pascal **move** procedure to manipulate the strings is a simple way to ensure the placement of substrings at specific character (= column positions).

The first of **Candidate**'s scoring methods, **CheckElements** is now called. This method accepts as parameters the *ElementList* to use in the comparison, which in this case is **Elements**, and the variables into which the resulting scores must be placed. **CheckElements** performs the preliminary

score and essential element check, placing the results in this case into the variables **PreliminaryScore** and **EssentialsPresent**. The latter is a boolean indicating whether the essential elements specified by **Candidate** are were found in **Elements**. The results are converted to text form and moved to the output string.

If the essentials were present, then the main syntax check must be performed. The second of **Candidate**'s scoring methods, **CheckSyntax**, performs this task. The method has the same name as the main procedure but is invoked, like all methods, preceded by the object instance's name: **Candidate.CheckSyntax** - and therefore is quite distinct from the main procedure. The method accepts an **ElementList** input parameter, and parses **Candidate**'s syntax tree, returning the failure count (in this case into the variable **FailureScore**) and the syntax score or appropriateness measure (into the variable **MatchScore**). The results are placed in the output string, and the iteration ends by writing the output string to **output** and destroying **Candidate** (using it's **Done** method).

When all the candidates have been checked (that is, when the end of **Candidates** is reached), all files are closed and **Elements** is destroyed - the syntax check is over.

A1.1.8.2 Elements: *ElementList*

The first, and simpler, of the two main SYNTAX objects is **Elements** - which is an instance of the object type **ElementList**. The description of this type, or class, thus defines **Elements**'s behaviour (as well as that of temporary element strings created and used elsewhere in the program). **ElementList** is a descendant of **List**¹, the general linked-list class defined in the object library module, OBJLIB. Please refer to the OBJLIB section in this appendix for details of the **List** object - a short reminder only is given here. **ElementList** is defined in the same way as all **List** descendants are - a special **ListItem** (see OBJLIB) descendant object **ElementListItem** is first defined to hold each element, and then the **ElementList**'s **Item** method is changed (from the method inherited from **List**, which returns a **ListItem** pointer to the current item), adding a typecast so that the returned pointer is of type **ElemListItemPtr**. The effect of this arrangement is that an **ElementList** object is a linked list of items which have a custom structure, and that the method **ElementList.Item** returns a pointer to the current item in that list. Like all **Lists**, The position of the current item can be set to the start of the list (by calling the **SetToHead** method), and advanced to the next item (using **SetToNext**) at will. Together with the test method **AtTail**, which returns a boolean indicating whether the end of the list has been reached, these methods allow the list to be traversed in a simple "while" loop:

¹ Actually it descended from **TracedList**, but when the compiler symbol **Trace** is undefined as is assumed here, **TracedList** is simply a synonym for **List**. When **Trace** is on, however, **TracedList** is a **List** with extra features added that allow its contents to be displayed on screen. Redefining **TracedList** based on the symbol was not necessary, but saves compiled code space.

```

Elements.SetToHead;
Elements.SetToNext; {the head of a List is}
                    {always a null item}
while not Elements.AtTail do
begin
    anelement := Elements.Item^.e; {use the current}
                                {item in some way}

    Elements.SetToNext;
end

```

The above discussion of *ElementList* is common (but for the names used) to all descendents of *List* and will not be repeated when other such list objects are discussed. In all cases, if the descendant list type is called *XXXXList*, then custom item type is of the form *XXXXListItem*, and the new *XXXXList.Item* method returns a *XXXXListItemPtr* which is a pointer to *XXXXListItem*. The features peculiar to the descendant list are implemented in at least two places - in the custom data fields added to the descendant item object type, and in the way the item initialises itself - i.e. in the definition of the *XXXXList.Init* method. There may be other custom methods that add further special features.

In the case of *ElementList* then, the special features are as follows: The *ElementListItem* has one custom field, *e*, which is of type *elementtype* and thus can hold an element as read in from the element input file. The *ElementList.Init* method accepts the name of such a file as a parameter, and builds the list by constructing a new item for each element read from the file, and storing that element in the *e* field of the newly constructed item. In essence the file is simply copied into a list. A second initialising (constructor) method is defined, called *CopyInit*. It is used to construct a copy of an existing *ElementList* instance. The copy is however, not a full-blown duplicate, since it uses the original's data. Thus while it has its own, independent instance variables for keeping track of a current item, the items themselves are shared with the original. To all intents and purposes it is a separate list, but does not occupy as much space as a true copy would. The only difference from a programming viewpoint is that the original should not be destroyed while the copy remains in existence. The duplicate *ElementLists* are used to keep track of sub-segments of the element string in the parsing process.

A1.1.8.3 Candidate: *SyntaxSpec*

The second of the two main SYNTAX objects is *Candidate* - an instance of *SyntaxSpec*. *SyntaxSpec* objects are compound and complex, embodying the form of the syntax specification, and the function of the syntax checker parsing algorithm. It can be thought of as an active syntax spec - one that has the ability to answer questions about itself in relation to an input string. The questions are asked by calling one of *SyntaxSpec*'s two action methods - *CheckElements* or *CheckSyntax*.

As explained in section A1.1.8.1, these methods perform the non-syntactic (preliminary score and essential element check) and syntactic (syntax score and failure count) comparisons of the *ElementList* passed to them as a parameter with the syntax spec. These method's workings will be described in later sections, once the way in which the syntax spec is internally represented has been explained.

The two other methods available with a *SyntaxSpec* object are the conventional constructor and destructor methods **Init** and **Done**. It is **Init** which must read the syntax spec file and place the information it contains in a suitable data structure within the object. **Done** reclaims memory space occupied by the structure and by the other components of the object, thus destroying it.

The data structure that holds the syntax spec information mirrors the structure of the syntax spec itself (see section 6.1, The Syntax Specification). Two separate structures contain the information from the two separate syntax spec sections - the element specifications and the syntax tree. The element spec list is contained in a field of the *SyntaxSpec* object called **ElemSpecs**. **ElemSpecs** is itself an object - an instance of *ElemSpecList*. Besides implementing the form of a list, *ElemSpecList* contains some of the functionality necessary to compare the element specs with incoming elements, and also to read that part of a syntax spec file that pertains to a single element spec - thus shifting some of the responsibility for initialisation away from *SyntaxSpec.Init*. Unlike the element spec list, the syntax tree is not explicitly contained in a field of the *SyntaxSpec* and is not an object. It is represented by a set of linked (but otherwise independent) node data types. Additional links between the nodes and the items in **ElemSpecs** allow the tree to be accessed. Thus **ElemSpecs** provides the entire syntax spec representation, either as direct contents or through links to the tree structure.

The remaining component of the *SyntaxSpec* object is the **BirdName** field - a simple string that stores the name of the syntax spec - usually the name of the species in question.

The syntax tree and its objects, and the *ElemSpecList* object must now be explained. Thereafter, the *SyntaxSpec* methods will be returned to in detail.

A1.1.8.4 The syntax tree: *SynTreeNode* and *SynTreeNodeList*

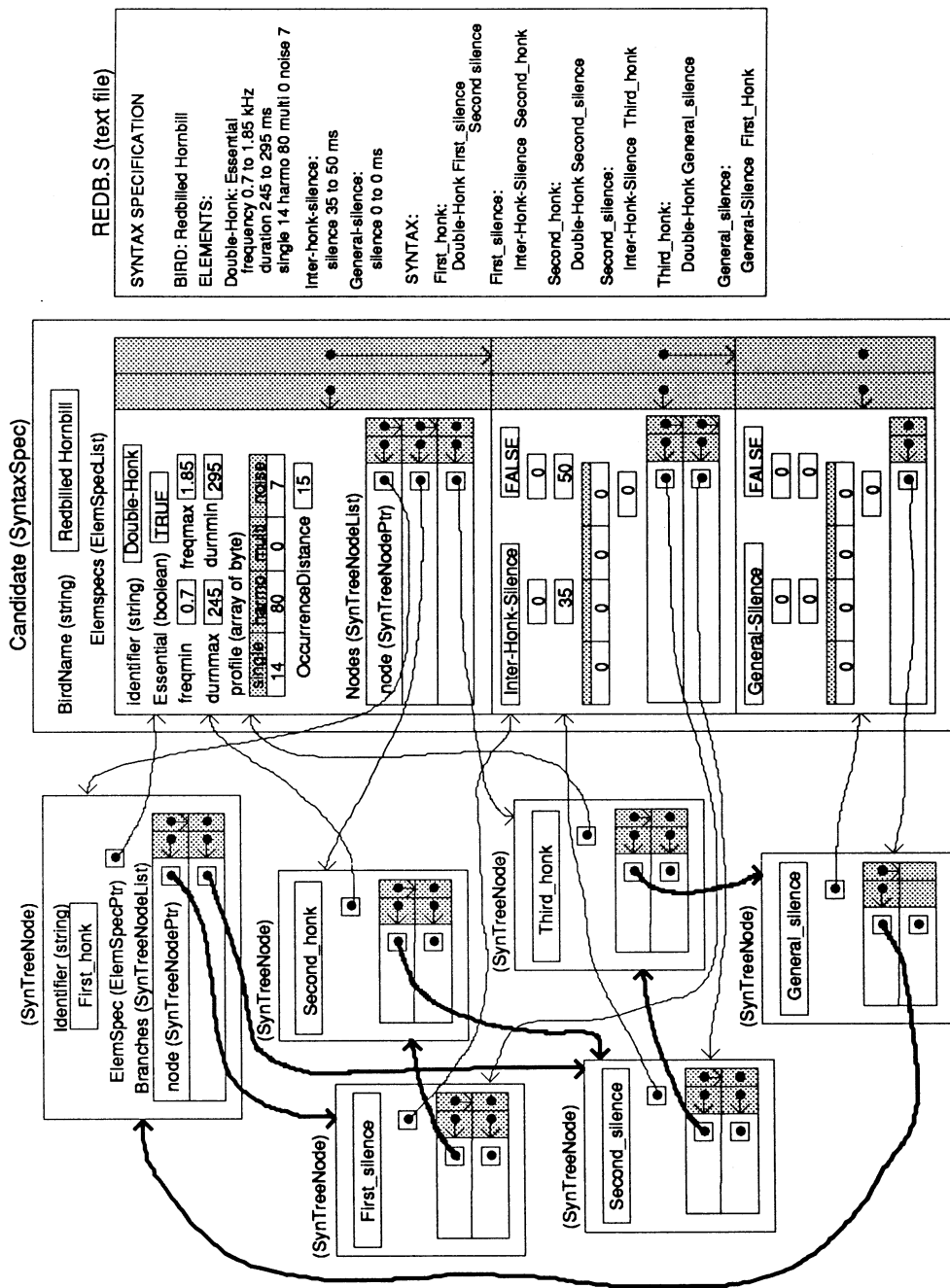
The entities used to represent a node in the syntax tree are not objects - they are passive data structures - records of type *SynTreeNode*. All the *SynTreeNodes* used in the tree are dynamically created and accessed through pointers - there are no named variables of this type. By considering a node specification in the syntax spec, it is easy to see what the characteristics of a *SynTreeNode* should be:

```
First_honk:
    Double-honk    First_silence    Second_Silence
```

A node must have a name (in this case, "First_honk"), it must be associated with an element spec (here, the element spec called "Double_honk") and it must have a list of branches - other syntax tree nodes to which it is linked (in this case, the nodes called "First_silence" and "Second_Silence").

The above characteristics are implemented by giving the **SynTreeNode** the appropriate data fields: **Identifier** is a string which can contain the name, **ElemSpec** is a **ElemSpecListItemPtr** - it points to an item in the list of element specs (which is the **ElemSpecs** field of the *SyntaxTree* object - see previous and following sections). The **Branches** field of **SynTreeNode** is a list - a simple list object of type *SynTreeNodeList*. The main custom feature of the latter list object is that it's items, objects of type *SynTreeNodeListItem*, have a pointer field (a **SynTreeNodePtr** called **node**) which can point to another **SynTreeNode**. Thus **Branches** is a list of pointers to other nodes in the tree.

Figure A1.5 depicts a syntax tree, with links between nodes shown by thick arrowed lines. It also depicts an element spec list, but this is discussed in the following section, so the links from it to the syntax spec, which are the thin arrowed lines, should be ignored for the moment. The syntax spec file from which the tree is built is also shown. The figure illustrates the nodes containing their name fields, their **ElemSpec** fields pointing into the **Candidate.ElemSpecs** list and their **Branches** lists containing small objects which have pointers to the other nodes - all as specified in the example syntax file.



REDB.S (text file)

SYNTAX SPECIFICATION

BIRD: Redbilled Hornbill

ELEMENTS:

- Double-Honk: Essential frequency 0.7 to 1.85 kHz duration 245 to 295 ms
- single 14 harmo 80 multi 0 noise 7

Inter-honk-silence:
silence 35 to 50 ms

General-silence:
silence 0 to 0 ms

SYNTAX:

First_honk:
Double-Honk First_silence Second_silence

First_silence:
Inter-Honk-Silence Second_honk

Second_honk:
Double-Honk Second_silence

Second_silence:
Inter-Honk-Silence Third_honk

Third_honk:
Double-Honk General_silence

General_silence:
General-Silence First_Honk

Figure A1.5 (previous page): How a syntax spec is stored in memory - a syntax tree composed of **SynTreeNode**s and an element spec list stored as **Candidate**'s **ElemSpecs** field. A file, object instance, record or simple variable is depicted as a rectangular block. The identifier of the data entity is shown outside the block, with its type in parentheses. The contents of the entity are shown inside the block, and may include sub-entities (fields) with their own names and contents. Arrays and lists (multi-element data entities) are depicted as touching blocks (one per element) joined into a single entity by a shaded bar. A list is distinguishable by the pointers within the shaded bar which link the items to each other and to their data. Pointer variables' contents are depicted by dots. If the pointer has a meaningful value, an arrow leads from the dot to the pointed-to data entity, otherwise the pointer is undefined or points to NIL. For conciseness the names of fields are not repeated in every place they occur.

The *SynTreeNodeList* object has a custom method, **Append** which allows the addition of the contents of another *SynTreeNodeList*. This ability is not used in the **Branches** of **SynTreeNodes**, but in the building of composite temporary branch lists in the parsing process (see later description of *EntryPointList*)¹.

A1.1.8.5 *SyntaxSpec.ElemSpecs: ElemSpecList*

The *ElemSpecList* is firstly a list of element specs, and the **ElemSpecs** field of the **Candidate** is an *ElemSpecList* that ultimately contains the information specified by the **ELEMENTS** section of the syntax spec. The list object itself has no special features. The item object, *ElemSpecListItem* naturally has the custom fields necessary to contain an element spec. These are: **identifier**, a string for the name of the elemt spec, **Essential**, a boolean reflecting whether or not the spec is essential, **freqmin**, **freqmax**, **durnmin** and **durnmax** for the characteristic frequency and duration ranges, and **profile** and **OccurrenceDistance**, the array of percentage values and the distance measure that specify the class occurrence profile range.

In addition to the element spec information fields, there are connections to the syntax tree, and methods concerned with matching elements and element specs. An *ElemSpecListItem* (i.e., an element spec object) can be interrogated as to whether it matches a given element, by calling the **Match** method. The method accepts an **elementtype** as input parameter, and returns a match level between 0 and 4, according to the scheme outlined in 6.2.3, Imperfect Elemets, and Scoring

¹ By rights the **Append** functionality should have been given to a new object, descended from *SynTreeNodeList*, so that its presence would not be able to interfere with the understanding of the object in the present context of its use in the syntax tree.

Mechanisms. The **SilentSpec** method returns a true or false value depending on whether the **ElemSpecListItem** represents a silent element spec or not. The **Nodes** field is a list of pointers to **SynTreeNode**s in the syntax tree - not surprisingly, it is a **SynTreeNodeList** object.

The reason an **ElemSpecListItem** needs to point into the tree is simple - it allows an efficient search for entry points. Recall that an entry point is a node in the syntax tree whose element spec matches an element at the start of a substring. When the parse is just beginning, or after a failure has occurred, a new subparse must begin at any node that matches the first element in the string (or after the failure). Since each node has an associated element spec, it would be possible to search all the nodes in the tree, comparing each spec until all matching nodes are found. This method would be inefficient, however, because there are always more tree nodes than element specs. By organising the element specs in a separate data structure (the linked list), while still maintaining the association of nodes to specs through the **ElemSpec** pointer in each node (see previous section), it becomes possible to search through the element specs independently of the tree, performing only as many comparisons as there are element specs. If in addition each spec "knows" which nodes are associated with it, the entry points can be deduced from such a search. This is the reason for the existence of the **ElemSpecList** as a separate object, and for the **Nodes** field in the **ElemSpecListItem**. Each item in the **Nodes** list points to one of the nodes that is associated with that spec in the tree. There is thus a two-way connection between tree nodes and element specs. Each node points to an element spec, and each spec points to all the nodes that point to it. An examination of Figure A1.5 should confirm this feature.

To complete the description of **ElemSpecListItem**, its **Init** method must be mentioned. This method takes some of the burden of interpreting the syntax spec file from **SyntaxSpec.Init**. It accepts a **SpecFileTextLines** parameter, **sf** (to be explained), a string (containing the name of the element spec) and a boolean (the essential flag value for the element spec), and fills in all the element spec fields of the newly created **ElemSpecListItem**. **SpecFileTextLines** is descended from the **TextLines** object defined in GENLIB. Please refer to the appropriate section later in this appendix for details - only a brief summary of relevant features appears here.

TextLines is an object that represents a text file with a line-by-line structure. When created, the name of the associated file is passed in its **Open** method, and from then on, access to the file need only be through the object. This access is by way of the **NextLine** method, which returns a string containing a line from the file. Successive calls of the method return successive lines. An important feature of the **TextLines** object is that it maintains a count of the lines read. **SpecFileTextLines** is a simple extension featuring an error message mechanism that includes the number of the line at which the error occurred. Calling the **Expected** method with a string parameter, say 'identifier', returns an error message string such 'Error in REDB.S, line 14: identifier expected' - where the name of the file (in this case REDB.S) and the line number are automatically inserted. By having

ElemSpecListItem.Init (and all other routines in SYNTAX) work with the *SpecFileTextLines* (which has already been opened as the appropriate syntax spec file) rather than directly with the text file, the details of file opening, file name handling and line count maintenance are hidden from the syntax checking code.

Returning to the *ElemSpecListItem.Init* method, we see that it can obtain the data for its fields by calling *sf.NextLine* and interpreting the resulting string. *Init* assumes that *sf* is open on a syntax spec file, and moreover that the current line is positioned at the part of an element spec following the title line. If the element spec in question is:

```
Element-A: essential
    frequency 0.1 to 0.3 kHz
    duration 100 to 150 ms
    single 10 harmonic 20 multiple 30 noise 40 %
    occurrence_distance 15 %
```

Then *Init* expects that the first call to *sf.NextLine* should return the string 'frequency 0.1 to 0.3 kHz'. If the wrong form is received, an error message is produced using *sf.Expected*. The algorithm proceeds with each line, checking for the expected keywords, producing error message if these are not found, or extracting the numerical information from the string otherwise. The numbers are stored in the appropriate *ElemSpecListItem* fields. Note that the *StringSeq* object, also defined in GENLIB, is used to access the strings. Just as *TextLines* apportions a text file into lines, so *StringSeq* apportions a string into words. Calling *StringSeq.GetWord* returns successive whitespace-separated groups of characters. See GENLIB for more details. This is suitable for the handling of element spec information, which is arranged as keywords and number values separated by spaces. *UpString*, too, is defined in GENLIB - it is a function which replaces all the characters in a string by their upper-case counterparts - and is used to make the functioning of *ElemSpecListItem.Init* case insensitive.

The final actions of *Init* are to initialise the *identifier* and *Essential* fields from the corresponding parameters (it doesn't read them from the syntax spec because they come from the first line of the element spec, which has already been read in *SyntaxSpec.Init* by the time it is invoked). Then it creates the *Nodes* list object, but cannot yet fill the list - this is done at a later stage by *SyntaxSpec.Init*, once it can see the overall structure of the tree.

A1.1.8.6 *SyntaxSpec.Init*

Init breaks its task into the two logically separate functions of reading in the element spec information and reading in the syntax tree information. These tasks are defined in separate sub-procedures, *CreateElemSpecs* and *CreateSyntaxTree*, which are simply called one after the other. Before doing so, *Init* initialises a local *SpecFileTextLines* object, *SpecFile*, with the syntax spec

file named by the passed-in parameter **CandidateName**. **SpecFile** is accessible by the sub-procedures, as is **seq**, a local *StringSeq* variable that is initialised and reused many times. As with *ElemSpecListItem*.Init, *SpecTextLines* and *StringSeq* objects are used to effect line- and word-oriented interpretation of the text in the syntax spec. Before calling **CreateElemSpecs**, the current line in **SpecFile** is advanced to the one after the 'ELEMENTS:' keyword. To do this, the **FindHeading** sub-function is used. **FindHeading** not only skips through **SpecFile** until a line beginning with a given word (the **heading** parameter) is found, but returns the remainder of that line (i.e with the the heading keyword removed). by calling "FindHeading ('BIRDNAME')"
Init thus obtains the text beside the 'BIRDNAME:' heading in the syntax spec, which is then stored in the *SyntaxSpec*'s **BirdName** field. Calling "FindHeading('ELEMENTS:')" is used only to position the current line in **SpecFile** (the remainder of the ELEMENTS: line is returned but not used). Note that **FindHeading** makes use of another sub-procedure, **SkipToHeading**, and the **seq** and **firstword** variables, to perform the actual scanning of the text file. This trio is used repeatedly in the Create... procedures in a standard way: **SkipToHeading** finds the next non-blank line in **SpecFile** and initialises **seq** to contain this line, and places the heading (first word) in **firstword**, leaving the current word (the next word returned by **seq**.Getword) at the second word in the line.

CreateElemSpecs

The **CreateElemSpecs** sub-procedure must initialise the **ElemSpecs** field of the *SyntaxSpec* by creating an *ElemSpecList* (by calling **ElemSpecs.Init** and then linking in new items as necessary) and reading the information under 'ELEMENTS:' in the syntax spec. The latter task is made simple by the fact that *ElemSpecListItem*.Init does the actual scanning of the element spec information, as described previously. All that is left for **CreateElemSpecs** to do is find the start of each element spec (by searching for headings terminating in a colon character). The heading (which is the element spec's name) and the word following it (which is ignored if it is not 'essential') are passed to *ElemSpecListItem*.Init method of a newly created *ElemSpecListItem*, which is then attached to **ElemSpecs** using its **Attach** method. Note that in the Turbo Pascal syntax, the creation, initialisation and use (in this case, attachment) of an object can all take place in a single concise step:

```
ElemSpecs.Attach(new(ElemSpecListItemPtr,
                    Init(SpecFile,name,essential)))
```

CreateElemSpecs stops looking for element specs when the 'SYNTAX:' keyword is encountered, leaving the **SpecFile** position in the appropriate place for the sub-procedure which is to be called immediately afterwards, **CreateSyntaxSpec**. Of course, if the end of **SpecFile** (indicated by the special *TextLines*.NextLine return value of '%END') is encountered without the word 'SYNTAX:' being found, the syntax spec file is not well-formed and **CreateElemSpecs** stops with an error message.

CreateSyntaxTree

Creation of the syntax tree is a two stage process. In the first stage, the syntax spec is read, node by node, and a new **SynTreeNode** is created for each node. The **SynTreeNodes** are, however not yet linked into the tree structure implied by the syntax spec - they are stored temporarily in linear list form. In the second stage, after all the nodes have been thus created, the links between nodes are made in accordance with the syntax spec, and the list structure abandoned. The reason for the two-stage process is simple: A node might need to be linked to (i.e. have as a branch) another node that has not yet been read from the file, and so does not exist in memory - there is as yet nowhere for the first node to point. The strategy is therefore to create a copy of the SYNTAX section in memory first, ignoring the links, then return and set the links.

In the first stage of processing, the syntax spec file is scanned in the same manner as it was for the element spec initialisation - titles ending with ':' indicate the start of a new node. A **SynTreeNode** is created, and passed to the **Init** method of a newly created **SynTreeNodeListItem**. The item is attached to the end of a temporary node list variable, **TempNodeList**. The first word on the line following the title is the name of the element spec associated with the node. The appropriate element spec in the **ElemSpecs** list is located by comparing with the **Identifier** fields of the items in **ElemSpecs**. The found item's location (a pointer to the item) is stored in the new **SynTreeNode**'s **ElemSpec** pointer field. At the same time a link in the other direction is established - a pointer to the new **SynTreeNode** is stored in a new **SynTreeNodeListItem** and attached to the **ElemSpec**'s **Nodes** list, so that the element spec will be able to find all the syntax tree nodes associated with it. See previous discussions and Figure A1.5 which illustrates the links being discussed here. If the node is label node (i.e. it's name is '.'), there is no associated element spec and the **ElemSpec** pointer is set to NIL to signify this.

Once the element spec pointer has been installed, the node's branch names are read one by one from the remainder of the line. The new **SynTreeNode** has a **SynTreeNodeList** field, **Branches**, whose items must point to other nodes in the tree. Since not all the tree nodes are in existence until the first stage is over, the **Branches** items (after being created and attached) are temporarily made to point to strings containing the *names* of the branch node (as read in from the spec). Since the branch items' pointers are "naturally" **SynTreeNodePtrs**, a typecast must be used to let them point to identifier strings.

The second stage of processing returns to each **SynTreeNode** in **TempNodeList** in order to install the **Branches** pointers correctly. A node's **Branches** list is stepped through one item at a time.

Each item has a pointer named **node**¹ which temporarily points to the name of a node. This name must be replaced by the actual node address. To do so, the name (a pointer to which is transferred to the working variable **TempId**) must be compared with the names of the all syntax tree nodes. A copy of **TempNodeList** called **AnotherNodeList** is used for this comparison, since we want to "keep our place" in **TempNodeList**. Our place in **TempNodeList** is the current **SynTreeNode** being fixed. **AnotherNodeList** is a second "placekeeper", but no copy of the **SynTreeNodes** themselves are made. So when a node with the same name (**Identifier** field) as the one in **TempId** is found in **AnotherNodeList**, the address of that node is the right address to be used in replacing the value of the **Branches** item's **node** field, and this is duly done.

Once all the **Branches** items are pointing at the correct nodes, **TempNodeList** and **AnotherNodeList** can be destroyed. Remember that the lists contained items (**SynTreeNodeListItems**) which pointed to **SynTreeNodes** - the items are destroyed but the nodes themselves are not. The nodes will be destroyed by the *SyntaxSpec*'s **Done** method. The nodes are now accessible in the way that is required by the BCR system - through **ElemSpecs** and the **Branches** lists - not sequentially as with **TempNodeList**. Note also that after the **TempId** has been used in the comparison, the dynamic string variable it pointed to, which was created in the first stage, is destroyed.

A1.1.8.7 *SyntaxSpec*.CheckElements and *SyntaxSpec*.CheckSyntax

We now come to the methods of *SyntaxSpec* which perform the various syntax parsing and scoring functions. **CheckElements** has the fairly simple task of finding the best match between each of the elements in the input string (the passed-in **Elms** parameter) and any of the elements in the **ElemSpecs** list. The best score for each element is added to a total which is then divided by the number of elements to give an average score. This average is the preliminary score, returned in the **AvgElementScore** parameter. The algorithm merely steps through **Elms**, and for each item (element) calls the **Match** method of every **ElemSpecs** item which returns a score (see *ElemSpecList*). The largest value obtained for each element is added to the total.

CheckSyntax is also a simple algorithm, owing to the fact that most of the work is done by an object which has not yet been described - the *EntryPointList*. This is a descendant of a *SynTreeNodeList* which can find and store the entry points corresponding to a given element (using the *EntryPointList*.Init method) and can parse the syntax tree using a given element string, finding

¹ The full handle is **TempNodeList.Item^.node^.Branches.Item^.node** - in English this is "the **node** field of an item in the **Branches** list of the node pointed to by the **node** field of the current **TempNodeList** item"

which entry point gives rise to the longest segment (using the **SetToEndOfLongestSegment** method). All that remains for **CheckSyntax** to do is therefore to oversee the parsing: It resets the element list (**Elms**) at the start of the procedure, and then enters a loop.

In the loop, **Elms**, together with the *SyntaxSpec*'s **ElemSpecs** list (through which access to the whole syntax tree can be gained) is passed to the **Init** method of a local *EntryPointList* instance, **EntryPoints**. The initialised **EntryPoints** thus contains the entry points for the current item in **Elms**. **CheckSyntax** then calls **SetToEndOfLongestSegment**, resulting in the current **Elms** item being moved forward to the point at which a failure occurred. **CheckSyntax** counts the failure, and adds the score of the recently parsed segment into the total, then destroys **EntryPoints**. The loop now repeats, so that a new set of entry points is created, and another segment parsed. When the end of **Elms** is reached, the accumulated failure count and segment scores are returned as the final scores for the element string, in the parameters **Failures** and **MatchTotal** respectively. **CheckSyntax** must also note a failure when there are no entry points at all (in which case clearly no attempt is made to parse the tree - instead a new iteration is initiated, with the current element being advanced by one so that a new set of entry points is found). The description of the syntax checking software can now be completed by examining the actions of *EntryPointList*.

A1.1.8.8 *EntryPointList*

This object contains the code that actually compares segments of an element string with a syntax tree, in the **SetToEndOfLongestSegment** method. The data stored by the object is the list of syntax tree nodes (so *EntryPointList* is a descendant of *SynTreeNodeList*). The nodes are the entry points for an element - that is, they are the nodes of the tree whose element specs match the element passed to *EntryPointList.Init* in the parameter **EntryElement**. **Init** finds the entry points by stepping through the element spec list (passed to it as the *ElemSpecList Specs*). When an element spec is found which matches **EntryElement**, the **Nodes** list of that element spec is appended to the *EntryPointList* object. Recall that *ElemSpecListItem.Nodes* is also a *SynTreeNodeList*, a list of pointers to all the nodes in the syntax tree that use the element spec in question. The **Append** method is a custom feature of all *SynTreeNodeLists* that allows a copy of one such list to be tacked on to the end of another. An initialised *EntryPointList* is thus composed of a chain of copies of **Nodes** lists taken from appropriate items of an *ElemSpecList*.

SetToEndOfLongestSegment works by taking each entry point in turn and seeing how far along the element string it can get. The element string has been passed in as the variable parameter **ReturnElements**. The current item in **ReturnElements** is assumed to be the same element that was passed to **Init** - the element for which entry points have been generated. When the method returns, it will have set the current **ReturnElements** item to some element further down the string; as far down as is syntactically possible. The main section of **SetToEndOfLongestSegment** loops,

stepping through the entry point list. For each entry point, a local *ElementList* called **Segment** is reinitialised with a copy of **ReturnElements** using the **CopyInit** method. As explained previously, this method makes a copy which behaves to all intents and purposes like a separate list - it can have a different current item to the original - but should not be used to modify the list contents - only read them. Its use here is to reset **Segment**'s current item to the original element position recorded in **ReturnElements**.

Thus **Segment** starts off as an identical copy of **ReturnElements**. **Segment**'s function is to record a new position in the element string (**ReturnElements** stores the original position - that of the entry point element). The next step in the loop is to call the sub-procedure **FindEndOfSegment**. This routine parses the syntax tree using **Segment** as the element string input. It starts from the current **Segment** item (which always starts off equal to the original entry point element) and enters the syntax tree at the current entry point. It performs the parse, and returns when a failure occurs. After it has returned, **Segment**'s current item is at the position of the failed element, the number of elements parsed is in the **SegmentLength** variable, and the sum of match scores accumulated until that point is in the **ThisSegmentScore** variable. **SetToEndOfLongestSegment** now checks whether **SegmentLength** is greater than the length of the longest of segment previously found (**MaxSegmentLength**). If so, the value of **SegmentLength** becomes the new **MaxSegmentLength**, and **ThisSegmentScore** and the **Segment** current position are recorded in **SegmentScore** and **LongestSegment** respectively. The loop now repeats with the next entry point, attempting to find an even longer segment.

When all the entry points have been checked, the current position of **ReturnElements** is made equal to that of **LongestSegment**, and since **ReturnElements** and **SegmentScore** are variable parameters, their values are returned to the caller as the "best segment" result of the subparse.

FindEndOfSegment

FindEndOfSegment takes a single parameter, **SyntaxTreeNode**, the entry point into the syntax tree. It does, however, use and alter **Segment**, **ThisSegmentScore** and **SegmentLength**, all of which are declared in the outer scope - their use as globals rather than parameters is simply to avoid a confusing proliferation of variable names.

Before entering the parsing loop, **FindEndOfSegment** initialises the variables mentioned above. It is important to note that the element associated with **SyntaxTreeNode** (the entry point) must by definition match the current item in **Segment** - that item is the element which was current when **ReturnElements** was passed in, namely the one which was used to find the entry points, and therefore matches all the entry points to some degree. So the initial value of **SegmentLength** (the number of matched segments in the sub-parse) is 1. The initial value of **ThisSegmentScore** is likewise the match level obtained by comparing **SyntaxTreeNode** with **Segment.Item^**. The real

work begins by advancing the current **Segment** item by one (using **Segment.SetToNext**). **MaxMatch** keeps a record of the highest scoring branch off the current node, so that the best branch may be selected when there are more than one matching branch. When this value is 0, no branches matched, which means a failure. It is thus initialised to a non-zero value to begin the loop.

The parsing loop consists of the following actions for each **Segment** element (temporarily stored in the **Element** variable): Initialise the list of effective branches (**AllBranches**), handle each branch node in **AllBranches** either as a normal branch or as a label node, advance the current element and tree node position, destroy **AllBranches** and repeat. Do not advance, and abort the loop, when there is a failure (no branches match). If there are no failures the loop carries on until the end of **Segment**, which is the same as the end of **ReturnElements**, is reached.

AllBranches is called a list of effective branches because it can consist not only of the current node's actual branches (i.e. **SyntaxTreeNode^.Branches**), but also of the branch lists of any label nodes branches. To understand this, consider the way a branch is "handled" as mentioned above:

If a branch is a normal node, one which contains an element spec, then that spec is compared with **Element** and the resulting match level is compared with the current maximum level (**MaxMatch**). After all the branches have been checked in this way, **MaxMatch** will contain the highest match level and **Branch** will point to the node that achieved it.

If the branch is however a label node, it has no element spec and therefore no choice can be made between branches of the current node until the branches of the label node are checked too. Instead of using a recursive method of examining these sub-branches, they are simply added to the current branches list (using **AllBranches.Append**). If any of the sub-branches contain more label nodes, their branches too will be added to the list. A situation which would lead to an infinite loop of adding "effective" branches occurs when a label node leads to another label node which leads back the original label node. This situation can be prevented in the syntax spec simply by ensuring that all loopback paths to label nodes are interrupted by at least one non-label node. This should not limit the expressive power of the syntax spec (although this assertion has not been proved conclusively). In the event that a problematic loopback path is inadvertently installed in a syntax spec, **FindEndOfSegment** will detect the problem and abort with a run time error, due to the use of the **NodeSeen** sub-function.

NodeSeen accepts a **SynTreeNodeList** parameter (**STNList**). It checks to see if the current item in that list appears previously in the list and returns **TRUE** if so. When called with **AllBranches** in the manner done in **FindEndOfSegment**, it indicates whether the current branch (which is always a label node, or else the check is not performed), i.e. the current item in **AllBranches**, has already been added to **AllBranches** previously. A **TRUE** result is a sure indication of a circular "branch gathering spree".

The final action before re-iterating in the main **FindEndOfSegment** loop is to advance both the element (the current **Segment** item) and the node (by making **SyntaxTreeNode** point to **Max-Branch**'s node, the branch with the best score), add the branch's score to the **ThisSegmentScore** total, and increment **SegmentLength**.

A1.1.8.9 Debug, Track and Trace features

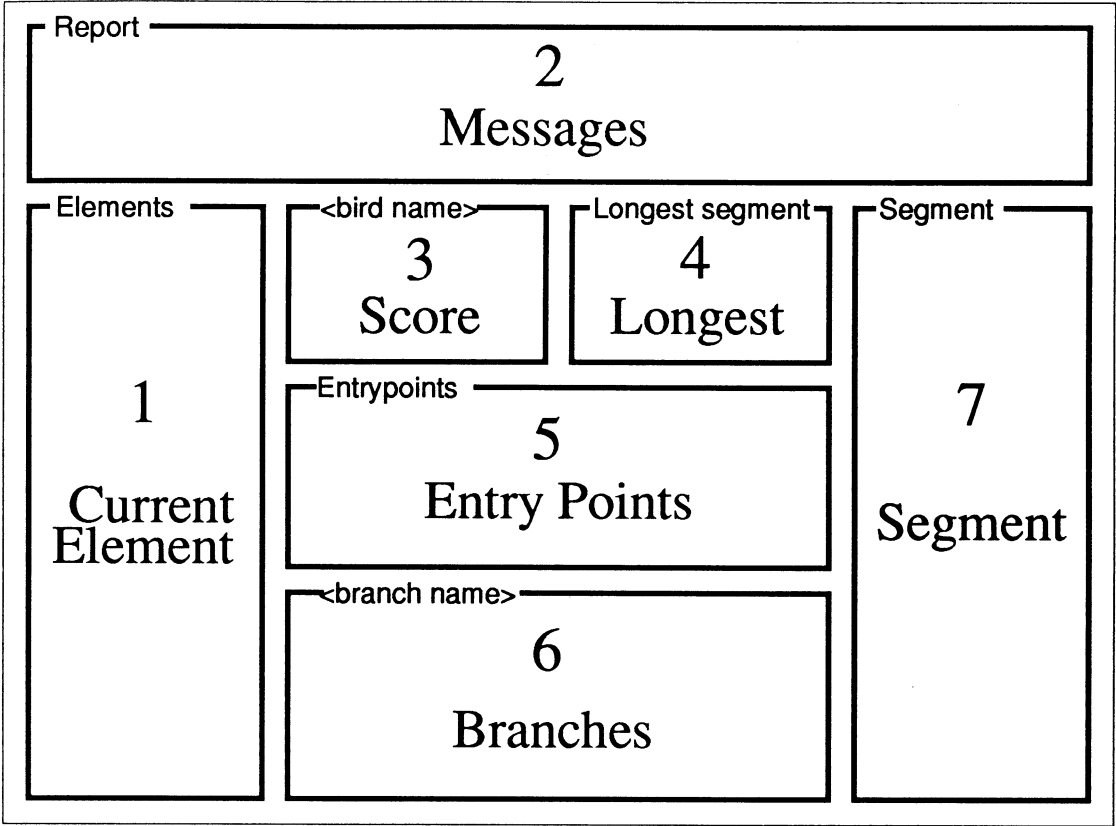
The diagnostic features mentioned in section A1.1.8.1 are now explained, in order to complete the picture of the behaviour of SYNTAX. As explained, the code for these features is scattered about, and is compiled only if the appropriate compiler symbols (**Debug**, **TrackResult** and/or **Trace**) are defined. The full appendix listing of the SYNTAX software therefore contains code of the following form, and it should be used as reference to the present discussion.

```
.
.
.   (* syntax checking code *)
.
.
{$ifdef Trace}
.
.   (* diagnostic and display code *)
.
{$endif}
.
.   (* more syntax checking code *)
.
```

Trace is the main feature, providing step-by-step on-screen views into the operation of the program. Track provides a subset of the Trace information, output to file as a permanent record of the parses performed. It is an alternate form of output from that produced by the **HumanResult** and **MachineResult** options described previously. Debug is used while developing the software, mainly to obtain feedback on the creation and destruction of objects. It has no use once the program is running without trouble. The coding for the diagnostic features will be discussed in general rather than covering each specific instance.

Understanding the Trace display

A number of text windows appear on screen when running SYNTAX with Trace. The following diagram identifies them:



Window 1 is the Current Element window, labelled "Elements". It displays the input element list, with the current element highlighted. Elements are identified by their class and their duration - the operator must refer to the full element string description (such as the output of the READELEM utility, see section A1.2.5) to see the other attributes of the element.

Window 2 is the Message window, labelled "Report". It contains various messages depending on the step most recently executed.

Window 3 is the Score window, and it is labelled with the bird name taken from the syntax spec currently being used. The score window shows the preliminary score, the syntactic match score (appropriateness score) and the syntactic failures score. For example if the three scores are 1.71, 20 and 16 respectively, window 2 shows "Prelim: 1.71" and "Syntax: M:20 F:16". The window is updated to reflect the latest score after all the entry points for the current element have been checked.

Window 4 is labelled "Longest Segment" and shows the length of the longest segment found to match the syntax spec from the current element. When there is more than one entry point for the current element, the one which results in the longest unbroken chain (segment) of matches before

a failure occurs determines the path chosen through the syntax spec. The last element in this longest segment is also displayed in window 4, although it can be difficult to determine exactly which element in the string it signifies.

Window 5 is the Entry Points window, labelled "Entrypoints". It shows the list of entry points, that is, the list of all the nodes in the syntax spec that match the current element. Nodes are identified by their name as specified in the syntax spec, with their associated element spec name being given in parentheses. The highlighted element in window 1 matches the element spec of all nodes in window 5.

Window 6 is the Branches window, and is labelled with the current entry point whose branches are being displayed. The highlighted node in window 5 initially has the branch nodes leading from it shown in window 6.

Window 7 is the Segment window, labelled "Segment". It shows the element string, identical to that of window 1, but with a later element highlighted. The sub-string between the current element (highlighted in window 1) and the element highlighted in window 7 is the *segment*. For each branch in window 6, the program checks the match between its element spec and the element currently highlighted in window 7. This match result is shown momentarily in window 8, the Match window.

As the operator steps through the parsing process by pressing a key, the following process takes place: For each current element (window 1), a set of entry points are found. For each entry point (window 5), a list of branches are shown. For each branch (window 6), the highlighted segment element (window 7) is checked, and a match score (window 8) shown. After all the branches have been compared with the segment element, the one that achieved the highest score is chosen. Its name now appears as the window 6 label, and the branches leading from it are displayed. This progress down the syntax tree, choosing the best scoring branch, continues, with the segment element being advanced each time, until a failure occurs (window 8 displays "FAILED").

The highlighted element in window 7 shows how far the parse progressed from the current element, and this segment is checked to see if it the longest one so far. If so, the length and the highlighted element name are displayed in window 4.

Now the next entry point is used to start the segment parse again. So the highlight in window 5 advances by one, and the branches of the new entry point appear in window 6. The segment element returns to its starting point, the element following the current one. Thus for each entry point, a segment is found.

Once all the entry points have been checked, the longest segment, as displayed in window 4, is chosen as the best path through the syntax spec. The current element is now advanced to the element at the end of this segment, i.e. the element at which the longest segment failed. The cumulative score of the matches in the longest segment is added to the match score in window 3, and the failure score is incremented. The process continues until the current element reaches the end of the string in window 1. If, for any current element, no entry points are found, the failure score is immediately incremented and the current element advanced by one. At each step an explanatory message is displayed in window 2.

A printout of the syntax spec and the element string is useful to have when tracing a parse.

Trace mechanisms

Code for the Trace feature consists of some special objects, some extensions of existing objects, and the trace statements placed in the active parts of the code, interrupting it to display the current state of the program on screen. All screen display is done using *TextWindow* objects, defined in the special library unit TEXTWIN (see appropriate section of this appendix).

The main purpose of the Trace feature is to display the state of the parse at various strategic points. In the SYNTAX software this state is denoted by the contents and state of a number of list object variables. An example is the **EntryPoints**, the *EntryPointList* instance declared by *Syntax-Spec.CheckSyntax*. Other examples found elsewhere in the code are the list of effective branches off the current node (**AllBranches**) and that containing the input element string (**Elements**). All the lists are descendants of *TracedList*, which with the Trace feature on, has custom methods that allow it to display itself.

TracedList.Show is the method which, when called with a *TextWindow* object passed as the **Win** parameter, causes the *TracedList*'s contents to be written inside that window. To achieve this, **Show** simply activates the **Win** window and then steps through the list, calling **ShowItem** for each item. Then **Win** is deactivated and **Show** returns. **ShowItem** is a virtual method - it does nothing as initially defined. The descendants of *TracedList* that wish to be displayable redefine their **ShowItem** method so that it does something useful. The simplest action for **ShowItem** is simply to write a line of text. Because every list has items of different types, and may wish to display different portions of these items, the definition of **ShowItem** is left to the different descendants in this way. For example, *EntryPointList.ShowItem* executes the following statement:

```
if ElemSpec <> NIL
  then write (ElemSpec.Identifier)
  else write 'LABEL');
```

The important contents of an entry point item are seen here to be the **Identifier** field, unless that item indicated a label node, in which case the word 'LABEL' is written. The behaviour of **ShowItem** here is thus completely specific to the particular list - another type of list might need completely different information to be displayed. All **ShowItem** methods must, however, produce some sort of screen output, usually a short piece of text. **Show** assumes that this output does not occupy more than a single line, (i.e. that a `writeln` statement is not used in **ShowItem**) and might produce untidy results if this is not so.

Show does not, however, rely on the descendant object to specify the format in which each item should be written. All lists have ordinary items, and a current item that has special significance. **Show** uses the **Win** attributes for all items, except for the current item, which is highlighted. The highlighting is done using the colours black and white for foreground and background in such a way as to make the text stand out against whatever the normal **Win** colours are. The overflowing of text is also handled by **Show** - when the window is full, the topmost items are scrolled upwards out of the window, and the latest items enter from the bottom of the window. If however, the current item appears in the window, **Show** stops the display of further items. The result is that **Show** displays the entire list, unless it is longer than the window, in which case it is scrolled out until the highlighted current item is visible at the bottom.

The list object types which are displayable (define **ShowItem** methods) are as follows (with the particular type of information used to represent an item given in parentheses):

- **ElementList** (the class of the element)
- **SynTreeNodeList** (the name of the element spec associated with the node)
- **EntryPointList** (the name of entry point's associated element spec, or 'LABEL')

The instances of these objects which are displayed by the program, each in a separate window (using the windows as screen-division objects) are as follows (with the information that they supply in parentheses):

- **Elems** (The element string and the element for which the current entry points were generated).
- **EntryPoints** (The entry points and the one currently being checked).
- **Segment** (The segment of the element string parsed so far - corresponding to the current entry point being checked).
- **LongestSegment** (The longest segment found since the checking of entry points began)
- **AllBranches** (The branches of the current tree node, and the one being processed. Only the actual branches, not the effective ones).

The various instances and the code which coordinates their appearance on the screen can be found in *EntryPointList.SetToEndOfLongestSegment* and *SyntaxSpec.CheckSyntax*. The code which displays *EntryPoints* is partly built into the *EntryPointList* object, which is not an ideal solution - it means that all instances of *EntryPointList* will display themselves. In the SYNTAX software there is only one instance, *EntryPoints*.

Besides the displayable lists, a number of other items of information are visible when tracing is enabled. A custom descendant of *TextWindow*, *ScoreObj*, is defined and instantiated as the global *Score* variable (declared before the *SyntaxSpec.CheckElements* method definition, so that it and *SyntaxSpec.CheckSyntax* can access it as can the main *CheckSyntax* procedure). The *ScoreObj* has two custom methods: *ShowPrelim* displays the passed-in number on the first line of the window, with legend 'Prelim:', and *ShowSyntax* displays the two passed-in numbers on the second line, as follows

```
'Syntax: M:<first number> F:<second number>'
```

The numbers are meant to be the preliminary score for *ShowPrelim* and the Match (syntax score) and Failure scores for *ShowSyntax*. *Score*, initialised in the main *CheckSyntax* procedure, is thus a permanent *TextWindow* for the display of the current scores.

Finally two pop-up windows report on transient events. The first is *MatchWin* in the sub-procedure *FindEndOfSegment* under *EntryPointList.SetToEndOfLongestSegment*. It is used to show the match score resulting from the comparison of the current element with the current syntax tree node. The second is the report window, which is used in many places throughout the code, through a call to the global *Report* procedure.

Report simply creates a two-line text window at the top of the screen, displays the message passed to it as a string, and waits for a key to be pressed before destroying the window and returning. Every time it is called it therefore suspends activity until the user has had time to read the message and wishes to continue. Calls to *Report* bring about the stepping action of the trace. Each time a new element is parsed, or any other significant event is about to take place, *Report* stops the action with a message explaining the impending event. Without the stepping action the displayed events would in most cases happen too quickly to be of any use. The user presses a key when ready for each step, the report window disappears leaving the screen as it was before, and activity resumes.

Debug

The Debug feature is an extension of Trace to include events that are of importance to the program developer rather than the BCR system user. The extra events are almost all related to the destruction of the various objects in the system. With Debug switched on, additions are made to the destructor (**Done**) methods of various objects, such as *SyntaxSpec*. The list item objects (*XXXXListItem*), which ordinarily inherit their **Done** methods from *TracedList*, define custom **Done** methods when Debug is on.

The extra code added to destructors is a call to **Report**, with a message explaining what is currently being destroyed. In the case of a list item, the name or other identifying feature of the item is reported. The **Done** method of a list item is called automatically by the **Done** method of the associated list, so the effect of the report message is to ensure that no list item is destroyed without the programmer's knowledge. In this way the programmer can ensure that the lists are created and disposed of in the manner intended. This validation was especially useful before the more comprehensive Trace feature was implemented, when calls to **Report** were replaced by **writeln** statements. Other places where Debug statements are added are in the main algorithmic sections (*EntryPointList.SetToEndOfLongestSegment*, *SyntaxSpec.CheckSyntax*, *CheckSyntax*) where they warn of an impending destruction of a list or other compound object. The programmer thus sees an announcement of the object to be destroyed, and then the name of each component item in the compound object, due to the debugging statements in the items' **Done** methods.

Track

Track (symbol **TrackResult**) is a separate feature from Trace - it can be used without Trace being on. The purpose of tracking is to produce a log or record of the parse. This log is sent to the standard output device. It replaces the result tables produced when either **MachineResult** or **HumanResult** (see later) are defined. The track records which nodes matched each successive element in the input string, the match levels of these matches, the positions where failures occurred, and the final total match score and failure count of the parse. The output contains the record of each candidate syntax spec's comparison with the input string, as specified in the candidate file.

Tracking is done using a specially defined object, *SegmentTrack*. This is a descendant of *Syn-TreeNodeList*, although in fact it could have been based directly on *List* since it does not use the former's custom **Append** method and its item object, *SegmentTrackItem* is descended directly from *ListItem*. *SegmentTrack*'s custom feature is its **Output** method, which dumps the list's contents to a specified text file. The *SegmentTrackItems* have two custom fields: **NodeId** is a string intended to hold the name of a syntax tree node, and **Score** can contain a match level result (integer number between 0 and 4).

The object is used by *EntryPointList.SetToEndOfLongestSegment* as follows: Two *SegmentTracks* are created, and pointed to by variables *CurrentTrack* and *LongestTrack*. The first of these is used in *FindEndOfSegment* to record the name of the matching node and the match score of each element in a segment. As the algorithm decides on a best match for a particular element, so a new *SegmentTrackItem* is made, initialised with the relevant information, and attached to *CurrentTrack*. Once a failure occurs, the segment is tested to see if it longer than the current longest segment. If so, then besides the other actions that take place (see previous *EntryPointList* description), the *LongestTrack* pointer is made to point to *CurrentTrack*, which frees the latter to point to the *SegmentTrack* that *LongestTrack* was previously referencing. Thus *LongestTrack* stores the track of the longest segment found, while *CurrentTrack* can be destroyed and used again in tracking the next segment.

Once the longest segment has been found, the track is sent to the track file with a call to the *LongestTrack.Output*. The *Output* method prints the node names and scores on a separate line, beside the elements which they match. In order to be able to do this, it must have access to the segment (part of the element string), which is the reason that an *ElementList* and a pointer to an *ElemListItem* are passed as parameters. The element list in question is *LongestSegment*, and the pointer (*ReturnElements*'s current item) shows *Output* where in the element list the segment begins (since a segment is in fact the full element list, only a part of which is meant to be used).

The above code takes care of the recording of each segment. The overseeing code, *Syntax-Spec.CheckSyntax*, takes care of recording the failures - by adding a line with a '>' symbol each time one occurs. The main procedure, *CheckSyntax*, places text in the track file to help distinguish between tracks made by different candidates, and records the final score information at the end of each candidate's track. Below is an example track made in a comparison of a Redbilled Hornbill call to a Redbilled Hornbill syntax spec, as found in Appendix F. Note that successive lines containing class/duration/node/match-level information correspond to successive elements in the input string.

Track/s of call in REDB1A.E¹ compared with syntax specifications in TESTSET²:

Syntax spec: REDB.S³ (Redbilled Hornbill)

```
-----  
| >4  
| MIXED 129.675 --- no match  
| >  
| SILENT 5.0 --- IDLE: 46  
| >  
| SINGLE 4.99 --- no match  
| >  
| SILENT 39.9 --- IDLE: 4  
| MIXED 259.3 --- FIRST_HONK: 4  
| SILENT 44.9 --- FIRST_SILENCE: 4  
| MIXED 259.3 --- SECOND_HONK: 4  
| SILENT 5.0 --- SECOND_SILENCE: 1  
| >  
| SINGLE 14.96 --- no match  
| >  
| SILENT 10.0 --- IDLE: 4  
| MIXED 284.3 --- FIRST_HONK: 4  
| SILENT 10.0 --- FIRST_SILENCE: 1  
| >  
| MIXED 129.67 --- no match  
|  
| Score for REDB.S:  
| Failures 7  
| Match total 30  
-----
```

1 Name of input element file containing Redbilled Hornbill call.

2 Name of candidate file (containing only one candidate, REDB.S)

3 Name of candidate syntax spec file

4 Symbol indicating a failure point in the element string

5 The first element in the string - MIXED is the class, 129.67 is the duration

6 The matching node name is 'IDLE:' and the match level was 4

A1.2 Support tools and utilities

A1.2.1 DRAWER and its support units ICON, MOUSE, PRTEGA

DRAWER is a patchwork of separate routines concerned with the display of the data produced by the various stages of the BCR system. The routines are linked by a common display coordinate system, and by their incorporation in a mouse-driven, menu-based user interface structure. The DRAWER software allows the user to select data for display or printing, and also has some data analysis functions and a portion which obtains input for use in the BCR pipeline. It is, however, by no means a comprehensive or integrated solution to the problem of controlling BCR experiments. It is a working tool, quickly developed and expanded as required, and not intended as a finished product for general use.

The functionality of DRAWER as a user interface is provided wholly by the ICON and MOUSE modules - general menu and mouse toolboxes. The ability to make a hard copy (printout) of the 16-colour display is provided another general tool, the PRTEGA module. These tool modules do not contain code that is in any way specific to the BCR system, nor are their functions used in the core recognition software. The tools and the programs that use them (DRAWER and LINKPEAK) are simply utilities that allow peripheral manipulation or organization of the BCR system data. These programs are not building blocks for further research or development of the software system. Therefore, their inner workings are of little importance to matter of bird call recognition. Instead of supplying a description of their workings, therefore, it is more important to describe how to use them.

In the case of DRAWER, using the software amounts to running it (rather than using it to build other software). The discussion will therefore concentrate on what can be done with DRAWER and how to do it. With the tools, usage implies incorporation into application software (such as DRAWER), so the functions of the tools and rules for using them in a program will be outlined. Although as stated the workings of the DRAWER code will not be explained, an understanding of the ICON and PRTEGA tools as presented here will facilitate the interpretation of the DRAWER code if this is desired.

A1.2.1.1 MOUSE

MOUSE consists of a set of procedures which are made available to the software that uses the MOUSE unit. The procedures perform the most common mouse functions, and are often simply calls to the mouse interface routines provided in the mouse driver through DOS interrupt 33h.

ResetMouse must be called before attempting to use the mouse. After it has been called, the **mousepresent** global may be examined - if TRUE, the mouse driver is installed and the mouse may be used. If FALSE, the mouse may not be assumed to be working as it should.

ShowMouseCursor and **HideMouseCursor** are used to display the driver-provided mouse pointer on the screen. The mouse can be used without the mouse cursor being displayed, but the user will not automatically be able to see where he/she is pointing, and such feedback will be up to the using software to provide explicitly.

GetMouseStatus is the basic workhorse procedure for using the mouse. Its parameters are variables into which the mouse's pixel/character position on the screen (**x,y**) and the status of the left and right buttons on the mouse (**left** and **right**) at the time that **GetMouseStatus** is called, are placed.

GetMouseMotion returns in parameters (**x,y**) the amount by which the mouse has moved since the last call to this procedure.

GetMousePresses returns the mouse position (**x,y**) when a particular button was last pressed, as well as the number of times that button was pressed (**presses**) since **GetMousePresses** was last called. The button in question is specified by the number 0 (left) or 1 (right) passed in the **button** parameter.

GetMouseReleases works exactly like **GetMousePresses** except that it is button *releases* that is important, not presses. A *press* is when a button is pressed down but not released. A click is a press followed by a release.

SetMouseHardCursor relates to the use of the mouse in text mode - a specially-provided cursor character (a background square) can be used, or the normal text mode blinking cursor (the hard cursor) can be moved around in response to the mouse movement.

MouseAction works analogously to the Turbo Pascal **keypressed** function - it is true if some mouse activity has occurred since the last check - either the cursor has been moved since the last call to **MouseAction** or **GetMouseMotion**, or a button is currently being pressed.

The remaining procedures are concerned with a simple interface to a feature provide by the mouse driver - the ability to install an interrupt procedure to be called every time any activity of the mouse occurs. Normally installing such a procedure is done at assembly language level, but **MOUSE** allows this to be done at as high a level as possible, using allowing a Turbo Pascal **interrupt** procedure to be specified as the mouse interrupt handler. Certain mismatches between the way the driver and Turbo Pascal preserve vital registers mean that **MOUSE** has to do some manoeuvring to achieve this. The details are technical, and the occasion for use of this feature is rare (it is not used by **DRAWER** or elsewhere in the BCR system) so it will not be further described. The procedures involved are **SetMouseInterrupt**, **GetRegs**, **PushDseg**, **PopDseg** and **RestoreTurboDseg** and their use is demonstrated in comment text in the source code.

A1.2.1.2 ICON

ICON provides a set of high level facilities for the quick development of simple graphics-mode, menu-based interfaces for programs. It uses MOUSE to provide mouse control of all appropriate user input features. It also allows the same control to be exercised from the keyboard. The key features provided are: Easily defined and used menus which overlay the existing picture on the screen without destroying it, a graphics-mode text input line editor, and a routine for selecting a rectangular area on the screen.

A basic function provided by ICON is **GetGraphInput**. It is the basis for obtaining position and character information, and is used by the menu and rectangle selection procedures. It may also be called directly - it is often all that is needed to provide mouse-based input for a program, eliminating the necessity of accessing the lower-level MOUSE procedures explicitly.

When calling **GetGraphInput**, a position and a rectangle are specified in the appropriate parameters (**px**, **py**, **leftlimit**, **toplimit**, **rightlimit**, **bottomlimit**). The effect of calling the procedure is to pass control to the user. The mouse cursor (a custom cursor provided by ICON, not the default mouse cursor) appears on the screen at the position specified, and is free to move within the rectangle specified. The movement is caused either by corresponding movement of the mouse, or by repeated presses of the cursor keys. **GetGraphInput** returns in one of two ways. If the **waitforkey** parameter is FALSE, it returns as soon as the cursor is moved away from its current position, or a key or mouse button is pressed. If **waitforkey** is set to TRUE, cursor movement does not cause a return - only a button or key press does. Only the set of keys specified in the input parameter **InKeySet** cause a response - other keys are ignored. Whatever caused the return, the following information is passed back to the calling software: The position of the mouse cursor in (**px**,**py**) and the key pressed in **Inkey**. The mouse buttons are treated for this purpose as keys with the ASCII codes of 1 (left button) or 2 (right button). If the return was caused by mouse movement (with **waitforkey** FALSE), **Inkey** contains the ASCII code 0. One other feature provided is the size of jump made by the cursor in response to cursor key presses. The unshifted keypad keys (or the special cursor pad keys) cause the cursor to move by one pixel in the chosen direction. The shifted keypad keys (Num-Lock must be off for this to work) cause the cursor to move in bigger steps. The size of the step (in pixels) is given by the value passed in the **quickstep** parameter.

ChooseBox is built upon **GetGraphInput**, and allows the user to specify the position and size of a rectangular area on the screen. When it returns, the coordinates of the top left and bottom right corners of the rectangle chosen can be found in the variables passed as parameters (**x1box**, **y1box**) and (**x2box**, **y2box**). When initially **ChooseBox** is called, the values in these variables specify a box to start with. The box borders are drawn in both white and black lines on the screen and whatever picture was on the screen before the call remains undamaged by the box being drawn

over it. The mouse cursor is anchored at one of the corners of the box, and moving the mouse or pressing the cursor keys moves that corner, changing the size of the box. When the TAB key is pressed, or the two mouse buttons pressed alternately in succession (left then right, or right then left), the cursor switches to become anchored at the diagonally opposite corner. Once the user is satisfied with the box, he/she presses the ENTER key, or the left button twice in succession, and the procedure returns, with the new coordinates and with the **status** Boolean variable parameter containing TRUE. If the user decides not to complete the choice, the ESC key or the right button twice in succession may be pressed. The procedure then returns with no change to the values in the corner parameter variables, and with **status** containing FALSE. The ability to specify a starting box and to escape from the selection process mean that a default or current box can be maintained that the user need only change if necessary, otherwise deciding to use as is.

The menu system provided by ICON requires slightly more preparation for use than a simple call to a procedure, but it is designed so that a minimum of such preparation is necessary while providing adequate flexibility. The system centres around the **Menu** procedure, which, when called, places a menu on the screen and waits for the user to select one of the items it contains. Selection is once again implemented through **GetGraphInput** is thus either keyboard- or mouse-based. Using the mouse, the cursor is moved vertically to the desired item and the left button pressed to select it. The cursor cannot move out of the menu's area, and can move only slightly to the left or right (so that it can be moved away if it is obscuring item text). Using the keyboard, the up/down arrow keys move the cursor from item to item (the shifted keys move it one item at a time) and the ENTER key selects. The ESC key or the right mouse button cause a return without selection - signalling to the calling software that the user cancelled the whole operation. When **Menu** returns, the menu disappears leaving whatever was underneath intact. As will be seen, **Menu** does not necessarily return once an item's action has been performed, but remains active so that another selection can take place.

Menu accepts parameters specifying its position on the screen (which will be adjusted automatically to fit completely on the screen if necessary), the number of items (which may range from 1 to the value of the constant **MaxNumItems**) and 10 pairs of parameters specifying the menu items. The first parameter in a pair is a pointer to a string containing the text that is to appear to denote the item. The second parameter is the name of the procedure that **Menu** must call when that item is selected.

A menu is therefore implemented in steps. First, the action to be performed on selection of each item must be encapsulated in a single procedure. The procedure must have no parameters. This is due to a language limitation - Turbo Pascal allows a variable or parameter to be a procedure, but must know the format the procedure parameter list. In other words, the type of the procedure variable, like that of any other variable, must be known at compile time. The format of procedures

in a general purpose menu cannot be predicted, so it is left parameterless, and passing of information to the procedure must be done through global variables. The procedure itself must be a globally accessible one. The **Menu** system therefore imposes a flat, less-than-optimally secure structure on the code which uses it. This negative factor is offset by the speed and simplicity of development which the system provides.

The second step is the definition of the string containing the text of the item. A pointer to that string is used in the **Menu** call, rather than the string itself, so that the text of an item can be changed by the action of one of the menu items - for example, an item which says "Turn Insert ON" can be changed to "Turn Insert OFF" without having to restart the menu. Below is an example of the first two code segments required to support a single menu item, "DoThisItem X". The action of this item is to count how many times it has been selected from the menu, displaying this count in place of the 'X' in the item's text.

```
var Count: integer;

{The item text string}
const DoThisItemStr: string[14] = 'DoThisItem 0';

{The item action procedure}
procedure DoThisItem;
var countstr: string [3];
begin
    Count := Count+1;
    Str (Count, countstr);
    DoThisItemStr := 'DoThisItem'+countstr;
end;
```

Note the use of the typed constant syntax of Turbo Pascal to allow declaration and initialization of the text string variable in one step. The third and final step in using a menu is to make the call to **Menu**:

```

Menu (0,0,1,
      @DoThisItemStr, DoThisItem,
      nil, nil,
      nil, nil,
      nil, nil,
      nil, nil,
      nil, nil,
      nil, nil,
      nil, nil,
      nil, nil);

```

At this point a menu appears on the screen with top left corner positioned at screen coordinate (0,0) and containing 1 item, which has the text: 'DoThisItem 0'. Each time the item is selected, **DoThisItem** is called, which results in the variable **count** being incremented, and the **DoThisItemStr** variable changing accordingly. The menu is redrawn automatically after an item's procedure has been executed, so the latest item text is always visible. The **Menu** procedure returns (and the menu disappears) when ESC or the mouse right button is pressed.

Menu accepts 10 item parameter pairs. When there are less than 10 items, the value **nil** can be passed for the string pointer, and an empty procedure provided by **ICON**, **null**, can be passed as the action procedure for the item. In fact it does not matter what is passed, since only the first **numitems** (the third **Menu** parameter) items can be selected. Using **nil** and **null** make reading easier. Note the use of the Turbo Pascal '@' operator to obtain the address of (pointer to) a variable rather than referencing the variable itself.

Once the various action procedures have been defined as above, it is a simple matter to construct a comprehensive menu structure. The result is clean (although as stated, flat-structured) code without explicit **case** constructs or loops. Submenus can be implemented simply by calling **Menu** again from within an item's action procedure. For further flexibility, an action procedure can cause its **Menu** to return as soon as it (the action procedure) itself returns. This is done by simply setting the global flag **MenuQuitFlag** to TRUE within the action procedure. Thus a menu can be temporary pop-up list of alternatives, one of which must be chosen, or it can be a more permanent control panel from which user-driven events are initiated.

The last **ICON** facility to be discussed is the **Prompt** function. Calling this places a line editing window on the screen (again, whatever was underneath the window is preserved and reappears when the window closes). The window position is determined by the values of the **xprompt** and **yprompt** function parameters. The top line of the window contains a message, or prompt, usually informing the user of what input is expected from him/her. The message text is specified in the

Message parameter. The second (bottom) line contains more text and flashing text-style cursor. The text in the bottom line is specified in the **Default** parameter. The bottom line can be edited - characters can be erased and inserted, and the left/right arrow keys move the cursor. In addition, if the first key pressed after the window appears is not a control-type key, the default text disappears, so that new input can be typed immediately without having to manually erase the default. Pressing ENTER (or left mouse button) ends the input session, and **Prompt** returns the string that was last on the bottom line. If ESC (right mouse button) is pressed, the default string is returned and the **Escaped** variable parameter will contain TRUE (otherwise it contains FALSE).

The number of characters in the editable field is limited to the value of the **NumChars** parameter. Typing beyond this length has no effect. A special case occurs when **NumChars** is 1. In this case, the editable line is missing (there is no feedback of the character typed) and there is no need to press ENTER - **Prompt** returns with the character as soon as it is pressed. Thus **Prompt** can be used as a simple interrupting message - when the user has read the message, he/she confirms this with a keypress and it disappears. Or **Prompt** can be used for simple selection (such as the "Abort, Retry, Ignore" message of DOS) as well as more general string (e.g. file name) or number (string must be converted using the Turbo Pascal **Val** function) input.

A1.2.1.3 PRTEGA

The PRTEGA unit provides a single utility - a screen dump. The procedure **PrintEga** copies the graphics screen to paper, using one of three common printer types. Besides the ability to work with many printers, it is designed for EGA graphics - 16 colour high resolution. Unlike most commercially available screen dump utilities, each colour can be translated to a different shade of grey, so that images and colour-coded graphs and diagrams reproduce faithfully. This feature is advantageous in the production of sonagrams, and allows all pictures generated in the BCR system (by DRAWER) to become useful hard copy plots.

When calling **PrintEga**, parameters must be set to choose amongst many options. **DumpDestStr** tells **PrintEga** the destination file or device of the dump - for example 'LPT1' sends it straight to a printer connected on parallel port 1, while 'a:dump.prm' sends the data to a file on a floppy disk, to be stored for later printing, perhaps elsewhere. **PrinterPar** specifies the type of printer, and can take on one of three constant values: **Epson**, **LaserJet2** or **Postscript**.

Epson refers to Epson-compatible dot-matrix printers with graphics capability, which includes most dot matrix printers currently available. An example of a model which is known to work is the Epson LX-800 printer. **LaserJet2** applies to an HP LaserJet II laser printer (on which the software is known to work) or compatible - again many other laser printers accept HP PCL, the language of the LaserJet, and will therefore work with **PrintEga**. Finally **Postscript** covers the

wide range of printers and other document production systems which use Adobe's PostScript page description language. A well-known example is the Apple LaserWriter, and the routine has been tested with a NeXT 400 dpi laser printer.

The **PrintScalePar** parameter allows rudimentary scaling of the output and is designed primarily for use with the **Epson** option. When **PrintScalePar** set to **Normal**, the picture appears with the same aspect ratio as that of the screen, its size determined by the size of the component dots of the printer. Depending on the size of the picture, it may not fit completely on an 80-column dot matrix picture, although it should fit on a 132-column one. The **Squashed** option halves one dimension of the picture, so that it does fit an 80-column (A4-type) page, but is distorted. The distortion may not matter for a graph plot, but probably does for an image. An alternative to squashing may be used when 80-column dot-matrix printers are all that is available - see below. The **Stretched** option doubles the size of both the picture dimensions - it is simply an enlargement feature. **Normal** and **Stretched** both work for **LaserJet2** printers, but **Squashed** is ignored (a **Normal** dump is produced). **PrintScalePar** has no effect when a **Postscript** printer is being used. Because of the power of the PostScript language, it is not necessary. The user may manipulate the dump in many ways by dumping to file, and editing that file with a text editor before printing (PostScript is a high level interpreted language and as such is stored in a readable, English-like form). In particular, to change the dimensions of the picture, the user may edit the values to be found in the PostScript file in the following lines:

```
%%BoundingBox: 0 0 <width in points> <height in points>
```

and

```
/widthpic <width in millimeters > mm def  
/heightpic <height in millimeters> mm def
```

The words in angle brackets <> represent numbers that will be found in the file. The number next to `"/widthpic"` is the horizontal dimension of the picture in millimeters, and can be altered at will. Likewise the vertical dimension, beside `"/heightpic"`. The file may be then be sent as is to a PostScript printer. If the file is to be included as part of an enclosing document, in other words if it is to be treated as an Encapsulated PostScript File (EPSF), then the values in the `"%%BoundingBox"` line are best altered to be consistent with the new width and height values - they should specify the same distances, but in units of points, that is to say 1/72 of inch, or 2.835 times the millimeter values. With a very small amount of effort, the user can learn to alter other aspects of the picture, such as its position on the page, and even its orientation. The PostScript dumps are therefore the ones to use for preparation of publishing-quality documents.

The **PrintFormPar** parameter is another option design primarily for use with dot matrix printers: **Full** provides a complete dump, while **Split** dumps the picture in two halves which can later be joined physically. By splitting a large picture, an undistorted dump can be obtained using an 80-column dot matrix printer. **Split** also works with **LaserJet2**, but is ignored with **PostScript**.

Finally the (rectangular) area of the screen to be dumped can be specified using the coordinates of the pixels at the top left (parameters **x1** and **y1**) and bottom right (**x2,y2**) of the area.

Initially, the mapping of colours to shades of grey is one that is useful when the picture on the screen is being generated specifically for printing. The colour 0 maps to the shade of black, the colour 1 to shade of grey just before black, and so on, higher colour numbers corresponding to lighter greys, until colour 15, which is white. The mapping can be changed in a number of ways. The variable in which the map is held, **GreyPalette**, is available globally, and therefore can be directly manipulated. It is indexed by the colour number and stores the shade values (also numbers from 0 to 15, denoting black to white) as its contents.

Alternatively, two automatic map-choosing algorithms are available, through the **ChooseGreyPalette** procedure. The procedure takes a parameter, **GreyPaletteChoice**, which can take on one of two values, **AsQuantized** or **Differentiated**. **ChooseGreyPalette** makes its choice based on the actual colours present on the screen (as opposed to the colour numbers - it looks at the values in the EGA system's colour palette). It tries to approximate colours to grey shades based on their intensity. A bright green will map to a light shade, while a blue will map to a darker shade - the algorithm is empirically based. This mapping is useful for converting a colour image (photograph-like picture) to its nearest monochrome equivalent on paper. The **AsQuantized** option leaves the mapping as described. The **Differentiated** option first performs the described conversion to grey, and then examines the resulting mapping. Where a number of colours are mapped to the same shade of grey, they are separated out again, so that all colours will be differentiable when printed. This option is useful when image-like accuracy is not required, colours instead having coded significance or being used to distinguish between objects that are otherwise alike. An example is a graph with a number of different traces superimposed on the same set of axes, each trace being a different colour.

Finally, the **InvertGreyPalette** procedure may be called to invert a picture - dark shades becoming light shades and vice versa. This is useful because the background of a screen picture is often black and the conventional background for printed matter (except images) is thus white.

After **PrintEga** is called, a column on the left of the screen appears with a few words indicating that the printing is active. The column changes to reflect the part of the picture currently being printed. When printing is finished the column disappears and the screen remains as before. **PRTEGA** can be compiled so that if there is a problem sending to the specified destination, a message (using

the **Prompt** function from ICON) appears to this effect, or this error message feature can be left out. **PrintEga** can be used in any high resolution or 16-colour graphics mode, including the EGA monochrome 640x350 mode, the 16-colour 640x200 mode, and the 16-colour modes of VGA hardware.

A1.2.1.4 DRAWER

DRAWER is constructed as a module - a Turbo Pascal Unit - that can be used by other programs, but in fact is more like a main program itself. It provides only one externally accessible procedure, **Display**, which when run places the user in a menu driven graphics environment. When the topmost menu is exited from, **Display** returns. **Display**, like many BCR processes, accepts as parameters strings informing it which files it is required to deal with. The basic file prefix, e.g. 'REDB1' is passed as **CallName**, and then suffixes to that name are passed in the others: **SpecName** specifies the spectral frames, **SonaName** the sonagram storage file, **ClassName** the classified frames file, and **ElemName** the element file name. The names must be consistent - if **SpecName** is 'A' and **ClassName** is 'B' then the parameters for REDB1B.CF must be under the section for REDB1A.SF in the Bird Call Descriptor file for the REDB1 set of experiments. If there is no consistency in this regard when the time comes for **Display** to open the relevant file, the program stops with an appropriate error message. **Display**'s design as a procedure means that if desired a 'shell' program could be written to allow a user-friendly choice of displayable files (based on the contents of the Bird Call Descriptors) after which **Display** can be called without any danger of inconsistency.

The rest of this section is taken up by the explanation of DRAWER features from a user (experimenter's) point of view.

Sonagram, Next Page, Back 1/2, Overplot

The display environment centres around the sonagram. It is this plot that is first drawn when **Display** is called. It is the sonagram view which is used to navigate through the call, switching to other views once the desired portion of the call has been reached. The sonagram's frequency axis is aligned horizontally on the screen, frequency increasing from left to right. Although the normal way of looking at a sonagram is with a vertical frequency axis, the horizontal axis on a computer screen usually has many more pixels, allowing a greater frequency display resolution. Since the frequency range is fairly fixed (not expected to exceed 0-8kHz) and since the time range of the sonagram can be anything from a few milliseconds to a few minutes, it made sense to let the time axis extend off the screen, so that later parts of the call could be scrolled onscreen when desired, thus not limiting the time range of the display. And once the time axis had, in effect, infinite resolution, the higher resolution screen dimension was best utilised for the frequency axis - which is best viewed all at once rather than scrolled on and off screen in sections.

So the time axis of the sonagram is vertical, with time increasing from the top towards the bottom of the screen. The origin of the sonagram is therefore near the top left corner of the screen. The magnitudes of the frequencies in the sonagram are represented by the colours of the trace. Black is low, white is high. The colours are in general not readily interpreted, although they do show the magnitude contours, so that the peaks, valleys and slopes in the sonagram can be deduced. The colours are, however, chosen so that when the sonagram screen is printed (using the built-in screen dumping feature of **Display**), they translate to shades with the same meaning as in a conventional sonagram - darker shades representing higher magnitudes. A screen dump of **Display**'s sonagram view provides a standard sonagram plot, with a greater range and subtlety of shading than one made with analogue equipment.

Although any screen hardware may be used when running **Display**, colours are employed to good effect, so that an EGA (Enhanced Graphics Adapter) system is best. For the sonagram view only, there is a special hardware/software configuration which allows the correct shading to be viewed on-screen, without having to print. The configuration is as follows: An EGA graphics card installed in the computer, connected to a dual-frequency *monochrome* video monitor, with the **DRAWER** software compiled so that instead of automatically detecting the appropriate graphics mode, it forces the system into EGA 16-colour 640x200 mode¹. In this set-up, the EGA card "thinks" it is connected to an EGA monitor, but is displaying at a resolution compatible with a CGA monitor. Dual-frequency monitors are designed to accept CGA signals (as well as monochrome ones) and to translate the colours to shades of grey. The effect is one which is unobtainable with the (higher-resolution) standard EGA system because EGA does not have the ability to produce 16 different monochrome shades. The only other way to obtain this shaded monochrome display is to use a VGA system, in which case a different palette of colours must be used - not currently implemented but simple to do if required.

Returning to **Display** and the sonagram view, it is now possible to explain the behaviour of the program by considering all the various menu options. The main menu appears in conjunction with the sonagram view to begin with. The first two items are 'Next Page' and 'Back 1/2', which are concerned with displaying different portions of the call. 'Next Page' shows the next portion (starting where the current screen, or page, ends) and 'Back 1/2' shows the previous portion, but only going back one half-page. The latter feature allows the user to frame a particular part of the call in ways which would not be possible if only full-page scrolls occurred.

¹ To do this, the compiler symbol EGALO must be defined by inserting the following line (preferably near the start of the source code)

```
{ $define EGALO }
```

The next main menu item, 'Overplot', turns on overplotting. Whenever a new page, or some other view as described later, is displayed, the previous view is cleared first. When overplotting is on, this clearing is not done. Overplotting allows most views in the program to be superimposed, which is useful for comparison purposes. When overplotting is off, the word 'OFF' appears next to the item, so that the full text of the item is 'Overplot OFF'. The 'OFF' toggles to 'ON' when overplotting is on. The 'Sonagram' item is next, and is used to redraw the current sonagram page - or more usefully, to return to the sonagram view from one of the other views.

Peaks

The 'Peaks' item switches to the peak frames view. The portion of the call currently being displayed (e.g. as a sonagram) is replaced with the corresponding information from the peak frames file. Peaks are displayed in exactly the same coordinate system as the sonagram - all that changes is that extracted peaks are shown as (time, frequency) point plots rather than the shaded magnitudes of the sonagram view. By switching between the peaks and sonagram views, the manner in which patterns of magnitude in the spectral frames file were translated into peak positions can be observed. Figure A1.6 shows a sonagram view and the corresponding peaks view. Note that in this and other figures taken from the screen, the plots have been turned "right way up" - that is, the frequency is on the vertical axis and time is on the horizontal axis, as in conventional sonagram plots.

In the peaks view (or any other view), using 'Next Page' or 'Back 1/2' automatically switches to the sonagram view before performing the desired scrolling action. Thus the sonagram view is the reference point for the current position in the call, and all other views descend from it.

The 'Classes' item switches to the class frames view, in which the classes of frames are represented by symbolic traces plotted at the appropriated time and frequency points. Before covering this view, which has many optional formats, the 'Elements' item is discussed.

Elements

'Elements' brings up the element string view. The frames, grouped together by the BCR system into elements, and stored in the appropriate elements file, are displayed as blocks of colour. A block extends along the time axis from the first frame to the last frame in the element group. The extent of the block along the frequency axis is not relevant, but each element is made up of a "stack" (stacked along the frequency axis) of layers of different colour. The colour of a layer in the stack represents the content, in the element, of frames of a particular class. The higher the content, the thicker the layer will be. Thus if an element is made up of 50% single-class frames, and 50% harmonic-class frames, its block will have two layers of equal thickness, one layer being in the "single" colour, and the other in the "harmonic" colour. The element string is thus represented by a sequence (along the time axis) of different length (corresponding to element duration) and dif-

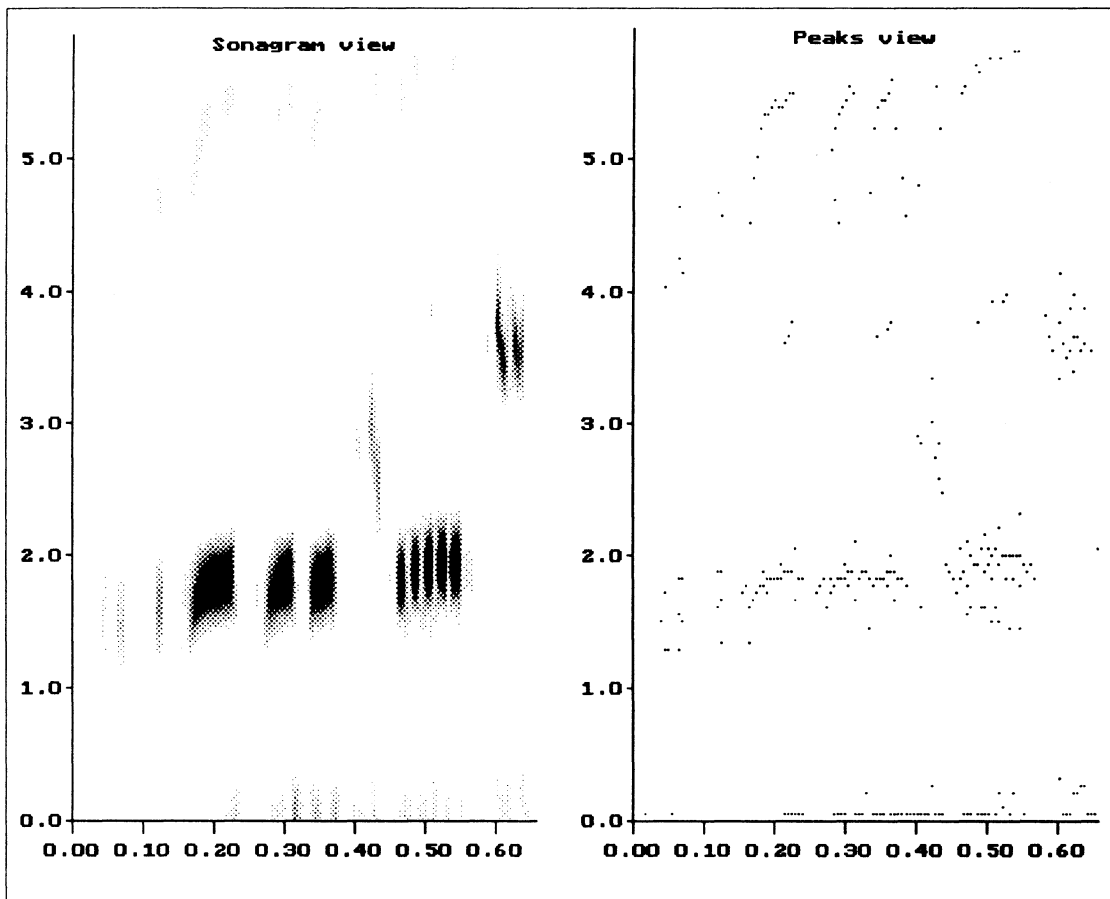


Figure A1.6 Left: A sonagram of portion of a call. **Right:** The peak frames view of this portion. A peak's position is represented by the coordinates (time, frequency) of a point. Magnitude of peaks is not represented.

ferently layered blocks. The blocks are positioned and sized to coincide in time with the frames (in the sonagram, peak frames or class frames view) which compose them. Silent elements are simply black blocks, seen as blank spaces in the string. Figure A1.7 illustrates.

In order to interpret the class composition of the blocks in the element view, a class-colour key can be obtained, as was done in the screen dump of Figure A1.7. The key is obtained by selecting the main menu item 'Functions' and then the 'Class key' item from the submenu that appears. This results in the cursor's appearance in the screen area. The cursor should be placed at the desired class key position, and the left button (or ENTER) pressed. The key is drawn and the function menu reappears. Escape from this menu to return to the main menu.

Classes

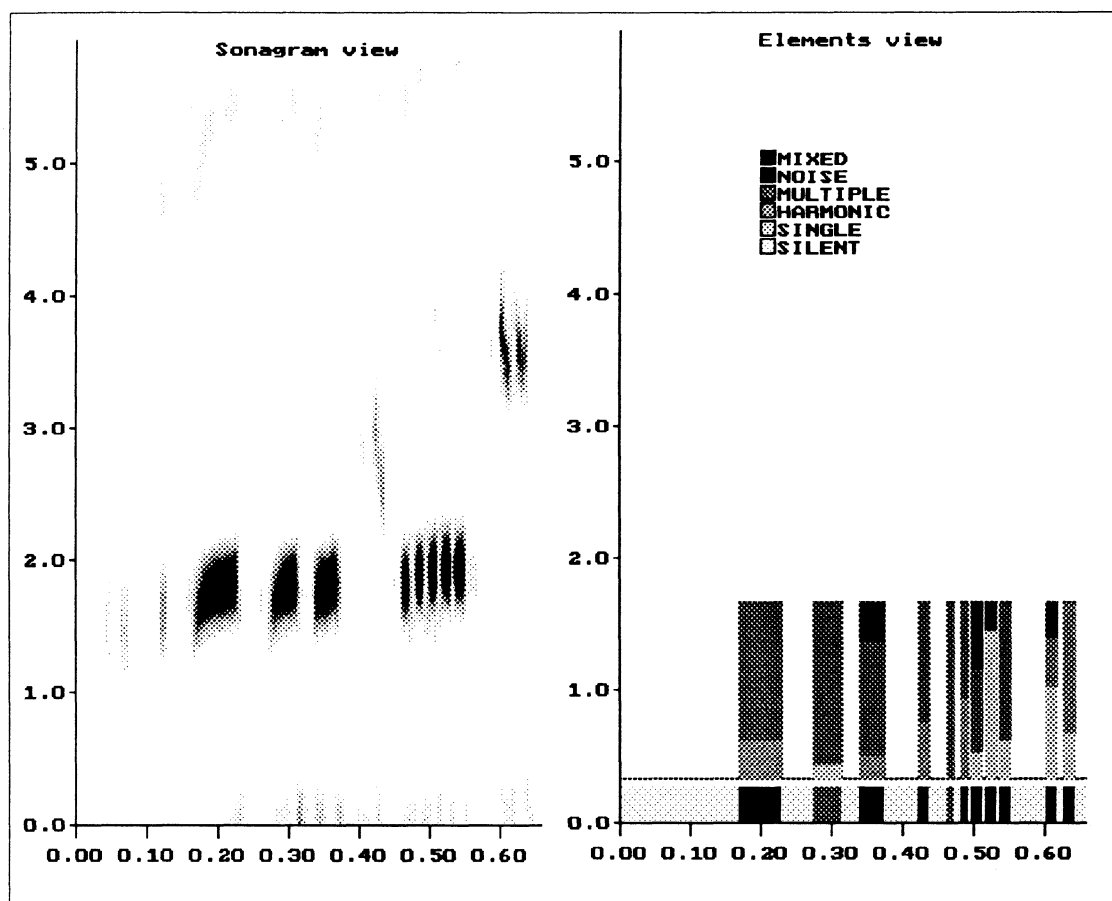


Figure A1.7 Left: A part of a sonagram of a call. **Right:** The elements view of this part. The time range over which a block extends indicate all the frames which are incorporated into a single element. The layers of colour in the block indicate the class composition of the element, according to the class-colour key shown.

The 'Classes' item on the main menu does not lead directly to a class-oriented view of the frames, but first offers a submenu of options. The various options cater for different representations, some more suited for screen than for paper, or vice versa, and some used in experimenting with hypotheses about how to detect element boundaries. Firstly, the submenu item 'Colour' toggles the use of colour to represent the class of a frame. Whether colour is currently being used is indicated by 'ON' or 'OFF' beside the word 'Colour'. When colour is off, white is used for all plotting. When on, the colours given by the class-colour key mentioned previously, are used.

The 'Frequency' submenu item plots a view in which each frame is represented by a horizontal line, the length of which is proportional to the characteristic frequency of the frame. The axes of

this plot are thus the same as the sonagram's, and the two views may be usefully compared. As mentioned the colour of a line shows how the BCR system classified the frame, as recorded in the class frames file.

The 'Magnitude' item is like the 'Frequency' option, except that the line length is now proportional to the characteristic magnitude of the frame. The horizontal axis is therefore different from that in the views encountered thus far.

The 'Reconstruct' item is similar to the 'Frequency' option, except that instead of (or as well as) colour, shape is used to represent the class of a frame. Instead of a line for each frame, a different shape is drawn depending on the class. This is useful for clarity in printed versions. For a 'single' frame, a dot is drawn. 'Multiple' is a short line. 'Noisy' is a long line, with a break in the middle. 'Harmonic' is a set of five short broken lines, spaced regularly apart from each other in the horizontal direction. The characteristic frequency, instead of being represented by line length, determines the horizontal (frequency axis) position of the shape. The centre of the shape, be it point, short line or long line, is aligned with the appropriate frequency value along the frequency axis. The exception is the 'Harmonic' shape, where the characteristic frequency determines the spacing between the five short lines. In this way, the information in the class frames (except for the magnitude) is represented in a way which symbolically reconstructs the shape of the corresponding sonagram. Figure A1.8 illustrates; remember that the horizontal direction referred to here is vertical in the figure.

The 'Product F x M' item plots the product of the characteristic frequency and magnitude of the frame, a way of combining the 'Frequency' and 'Magnitude' items - in the hope that this would offer insight into detecting the element boundaries, if they were coincident with sudden changes in either the frequency or magnitude, or both.

The 'Differential' submenu item toggles differential plotting. When 'OFF', the 'Frequency', 'Magnitude' and 'Product F x M' items behave as described, and the 'Operation' item has no effect. When 'ON', the values plotted (always using lines of horizontal length proportional to the value in question, and of colour representative of the frame's class) are calculated using both the current frame and the previous one. The calculation depends on the 'Operations' setting. If set to '-', the appropriate values are subtracted. If set to '+', the values are divided. Thus, with 'Differential ON' and 'Operation -', the 'Frequency' item plots the difference between adjacent frames' characteristic frequencies. These options were again intended to attempt to highlight element boundaries. The conclusion from this experimentation, as explained in the chapter on element discrimination, was that amongst the hypotheses tried, only the magnitude gave any useful indication of element boundaries.

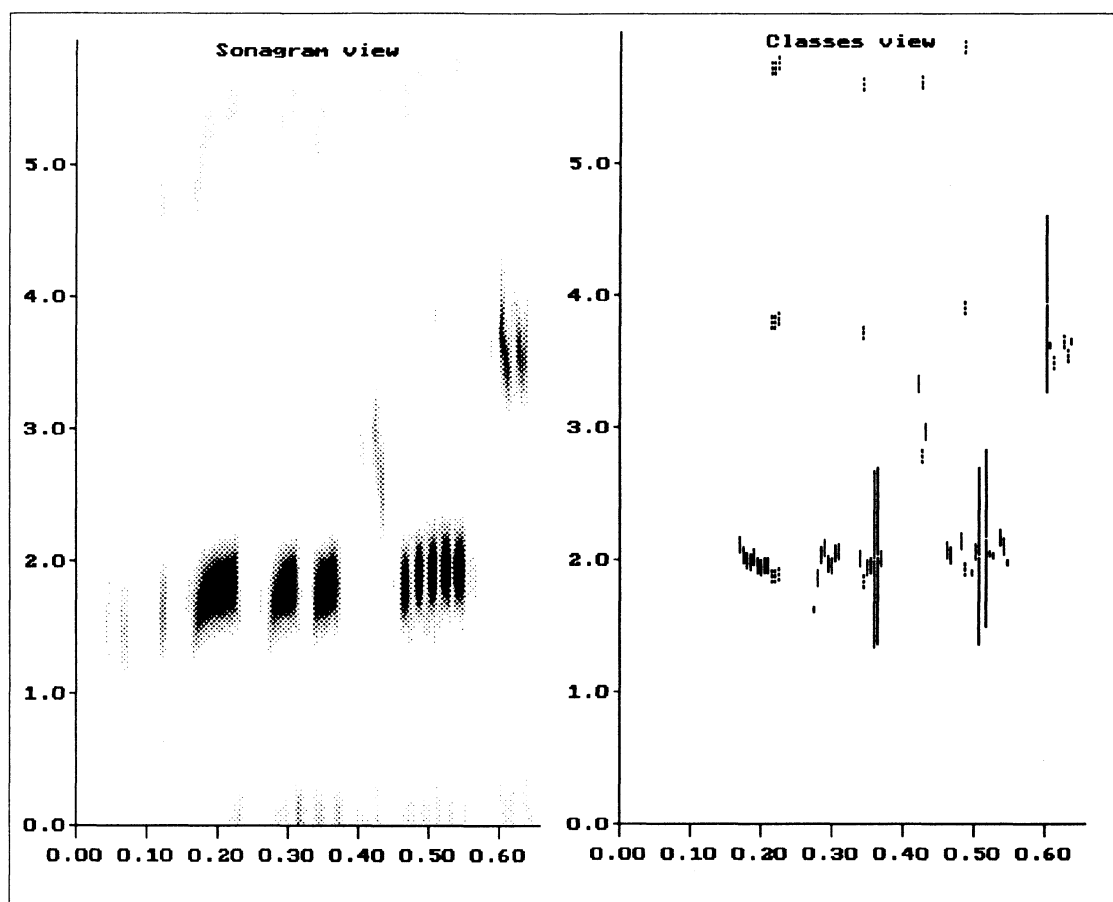


Figure A1.8 Left: Portion of a sonogram of a call. **Right:** A class frame view of the same portion, using the 'Reconstruct' option. See text for interpretation of the symbols

Completing the discussion of 'Classes' submenu, the 'Scale' submenu item prompts the user for a number, which is used to multiply all plotted values - useful for the magnitude and differential plots where the scale is not necessarily as well-adjusted as for the frequency plots.

Analysis

The next main menu item to be dealt with is 'Analysis'. This is once again concerned with testing element discrimination hypotheses by plotting various functions of the frames' parameters and looking for meaning in the resulting curves. The test performed is explained in section 5.2.1, and involves the use of a moving "window" of frames. The analysis consists of counting the proportion of a particular class of frames occurring in the window as it moves along the length (time axis) of the entire call.

Each analysis as performed in DRAWER can include a set of tests, with a different window length used in each test. The tests are performed as a group, increasing the window length by a fixed amount for each test. The result of each test (proportion of frames in window versus frame number (time) is plotted. The proportion is plotted on the horizontal axis at an arbitrary scale, and the time is plotted on the vertical axis at the same scale as for all other plots in DRAWER, so that the plots can be overlayed onto the call for comparison. Each plot in a set of tests is offset along the horizontal axis by fixed amount from the previous plot, so that they do not overlap and obliterate each other. The colour of the plot is determined by the class being analysed according to the class-colour key mentioned previously.

When 'Analysis' is selected it provides a submenu for controlling the parameters of the set of tests. The first submenu item is 'Class', which allows the class used in the analysis to be specified. The 'Mode' item allows the a choice between two plotting modes - 'Normal' plots the actual number of frames of the specified class found within the window, while 'Proportional' normalises this number by dividing by the window length. Next are 'From' and 'To', which specify the range over which the window length is to be varied. 'Jump' specifies the increment in window length to be used in successive tests. Thus if 'From' is 2 and 'To' is 22, with 'Jump' set to 4, then the analysis will consist of 6 plots, using window lengths of 2, 6, 10, 14, 18 and 22. 'Scale' allows a change of (horizontal) scale, and finally 'Analysis Plot' performs the analysis and results in the plot appearing.

The element analysis is designed to be performed so that the plot overlays other plots, allowing the experimenter to look for correlations between the analysis and the element boundaries. Figure 5.7 showed the plots for a particular class overlayed on a sonagram.

Functions

The last main menu item is 'Functions', which provides a submenu of miscellaneous facilities.

The first submenu item is 'Interval', which allows the user to choose any time interval in the plot. A submenu appears containing the three items 'Beginning', 'End' and 'Find Backgnd'. The interval is chosen by first moving (using 'Next Page' and 'Back 1/2' on the main menu) to the page where the start of the interval desired can be seen. Then 'Function' - 'Interval' - 'Beginning' menu sequence is selected. The cursor may then be pointed at the exact spot where the interval is to begin, and the left button (or ENTER) pressed. The page is redisplayed with all frames subsequent to the interval start point highlighted in reverse video (black becomes white, etc.). Now the user returns to the main menu and moves to the page where the interval end point is visible. Again entering the 'Interval' submenu, 'End' is selected. The cursor is pointed at the precise end point, and finally the page is redisplayed with only those frames between the beginning and end points highlighted.

If the start and end points are on the same page, then there is no need to exit from the 'Interval' submenu - simply choose the two points one after the other. To clear the interval, select a start point beyond the current end point.

The reason for selecting an interval is so that some function can be performed on that part of the call. Currently, only one such function exists, but more could be added in the same way. The function is the calculation of the background magnitude, which is then stored in the Bird Call Descriptor for use by the pipeline. The user selects a part of the call which reasonably represents the background - that is, a part where there are no significant elements. The interval may appear completely silent (but is not, since the sonagram has a magnitude cutoff mechanism, defined in the Bird Call Descriptor) or it may contain some spurious sounds that are present in many places in the call. Having selected this "representative background" interval, the user selects the 'Find Backgnd' option of the 'Interval' submenu, and the calculation is performed and written to the Bird Call Descriptor.

The second 'Function' submenu item is 'Dump screen'. This allows any view to be sent to the printer using the **PrintEga** routine from the PRTEGA module - please refer there for details of options. All the **PrintEga** options can be set from the submenu provided: Printer is set by toggling the 'Printer' item between 'HP LaserJet', 'Epson compat' (for Epson compatible printers) and 'PostScript'. PrintScale is set by toggling 'Scale' between 'Strtch' (for Stretch), 'Normal' (for Normal) and 'Squash' (for Squash). PrintForm is set by toggling 'Format' between 'Full' and 'Split'. After setting these three options, 'Begin Dump' can be selected. The user can then choose the portion of the screen to be dumped by adjusting a box - the cursor adjusts a corner of the box, TAB switches to the opposite corner, ENTER indicates when box is satisfactory (see **ChooseBox** procedure from ICON module). Finally the destination of the dump can be specified, it should be 'LPT1' or 'LPT2' to go directly to the printer, or a filename such as 'c:dump1.ps' to be stored in a file for later printing. After the destination has been entered, the dump begins, indicated by writing and a changing display in the leftmost column of the screen.

The third 'Functions' submenu item, 'Graph axes', allows a frequency axis, time axis and/or a title to be added to any view. A submenu offers various options. The 'Freq Axis' option toggles the plotting of a frequency axis 'ON' or 'OFF', so that one can either have an axis pair, or a time axis alone. The 'Time Axis' option toggles the calibration of the time axis: The numbers along the axis may indicate the absolute time (in ms) since the start of the call ('ABS' setting), or they may be relative to the start of the axis - so that the axis origin is marked 0.0 ('REL' setting).

The 'Draw Axes' item initiates the drawing of the axes once these settings have been chosen. The user is required to choose the place and dimension of the axis by manipulating a box on screen. The cursor changes the size of the box (which means the length of the axes). By switching corners

(using TAB key or two alternating mouse button clicks) the origin of the axis system can also be changed. The values displayed along the axis will be automatically correct, whatever the axis position. When the box is in the desired place, the user double clicks the left button (or pushes ENTER) and is prompted for a title (which may be left blank). The title be will written sideways, as is the axis text, and positioned along the right vertical box boundary, centred vertically. The plot is therefore "right way up" when the printed page is rotated 90 degrees.

The last 'Functions' item is 'Class key', which has already been covered, in the section on the 'Class' main menu item.

A1.2.2 Bird call reconstruction: SOUNDGEN, LINKPEAK

A1.2.2.1 SOUNDGEN

SOUNDGEN has nothing to do with the recognition part of the system, and is not involved in the conversion of (ultimately) real-time data from one form to another in a "production line" of processes. Rather it translates static data from a database (the data happen to be of the same form as the peak frames moving through the BCR pipeline) into a time waveform for playing back. SOUNDGEN is not concerned with the source of the data, however, or with the action of playback - it simply implements the waveform generation algorithm, outputting a file containing 12-bit samples.

Inputs

The main input is the file of Linked Peak Frames (.LPF), the output of PEAKPICK, which may have also passed through LINKPEAK. In the latter case, the link values¹ from each frame will have been meaningfully set, and in the former, the links will be off (i.e. the frames will be unlinked). SOUNDGEN uses the link values if available. SOUNDGEN generates frames of sampled time-waveform, so it must get the 'Sampling_frequency:' and 'Window_length:' from the Bird Call Descriptor. The 'Number_of_frames:' is read to specify the length of input and output files. In order to make maximum use of the range of quantised amplitude values available, SOUNDGEN scales the waveform it produces such that the maximum magnitude in the signal coincides with the maximum amplitude quantum level. To be able to do this it reads in 'Max_Magnitude:' from the Bird Call Descriptor.

Actions

To invoke the SOUNDGEN process, the **GenerateSounds** procedure is called.

The main loop of **GenerateSounds** consists of reading in a linked peak frame, generating a component sine wave segment (in **ComponentSeg**) for each peak in the frame using the **GenerateSine** procedure, adding together these component segments in the array **CompositeSeg**, and scaling this array to fit the quantisation range before writing it out as a frame in the Digital-to-Analogue Samples (.DAS) file. The latter is of the same format (simply a single sequence of 12-bit linear samples embedded in 16-bit integers) as the Analogue-to-Digital Samples (.ADS) file used as initial input to the BCR pipeline (see SAMPLE).

GenerateSine is capable of interpolating between two sine wave specifications. That is, given a starting magnitude and frequency pair, and an ending magnitude and frequency pair, it produces a set of samples of a segment of sinusoid, which has the specified magnitudes and frequencies at

¹ See below for discussion of the link value's function.

the start and end of the segment, and intermediate values in between. The segment starts at a phase angle specified by the value in the **PhaseAngle** variable parameter, and the angle at which it ends is passed back in this variable. In summary, the waveform generated is made up of samples from the following function:

$$A(t) \sin[\omega(t)t + P]$$

where

$$A(t) = \frac{A_2 - A_1}{T}t + A_1$$

$$\omega(t) = \frac{\omega_2 - \omega_1}{T}t + \omega_1$$

and A_1 and A_2 are **StartMag** and **EndMag**, ω_1 and ω_2 are $2\pi(\text{StartFreq})$ and $2\pi(\text{EndFreq})$, respectively, T is the time duration of the segment (equal to the product of the parameters **TimeStep** and **Samples**) and P is the passed-in value of **PhaseAngle**.

Thus, when **GenerateSounds** calls **GenerateSine** for a particular peak in a frame, it can specify different start and end frequencies and magnitudes. These values are taken from the values of two linked components in two successive frames. For example, having read in frame n , if peak 2 in frame $n-1$ is linked to peak 3 in frame n , then the starting magnitude and frequency will be the values taken from the **Freq** and **Mag** fields of the second element of the previous peak frame, and the end values will be those in the third element of the current frame. In that example, such a link would be indicated by the value of 2 in the **link** field of the current frame's third element. If there is no link, (if the **link** field of the current frame contains 0), the start and end values are equal, and taken from the current frame.

The link means that peaks which originate from a single "thread" of sound (say, a rising whistle) can be translated back to a continuously changing sinusoid, instead of abrupt jumps in the frequency and magnitude of the whistle that occur when there is no correlation mechanism between peaks. In effect, the discrete samples of the changing acoustic phenomena in the call are interpolated to reproduce a smooth time waveform. Of course, when there are sudden changes in the call which should not be smoothly joined, there should be no links between frames.

Subjective playback experiments would seem to indicate that the absence of links between peaks, even when those peaks should theoretically be linked, does not have a marked effect on the quality of sound produced. However the difference between linked and unlinked reconstructions can certainly be seen in the sonagrams of some calls, as Figure A1.9 shows. It is possible that the ear, as

opposed to the eye (viewing a sonagram) is not particularly sensitive to the kind of abrupt changes in magnitude and frequency that occur when links are absent. If this is the case, it should apply even more so to abrupt changes in the phase of the sinusoid between frames, because the ear is not sensitive to phase itself. However, the tests were by no means exhaustive, and the quality of sound reproduction equipment used low, so that the subtle effects of links and phase changes could have been masked by more gross errors.

Thus SOUNDGEN indeed makes provision for links, and likewise for ensuring a continuity in the phase between linked peaks. Incorrect phase could possibly affect the amplitude of the sound, when the individual component sinusoids are added to produce the composite result. The algorithm allows the phase to be preserved between linked peaks by passing the end phase of the sinusoidal segment for the peak in previous frame as the starting phase for the corresponding peak in the current frame. For unlinked peaks, the starting phase is set to zero.

As explained previously, the output waveform is scaled and quantized so that the sample value 2047 (the highest possible 12-bit sample value) results from the amplitude given by 'Max_magnitude:', the maximum magnitude previously found in the call and stored in the Bird Call Descriptor. This calculation is slightly erroneous, in that the 'Max_magnitude:' value applies to individual components in the frequency domain and not to the composite time waveform. It should thus be applied to the **ComponentSeg** values rather than the **CompositeSeg** ones. The reason for this error is historical and there is no good reason that the algorithm should not be changed. Its effect is to produce a waveform that does not, in fact, take full advantage of the quantization range, and should be corrected if quantization noise or distortion is found to be significant.

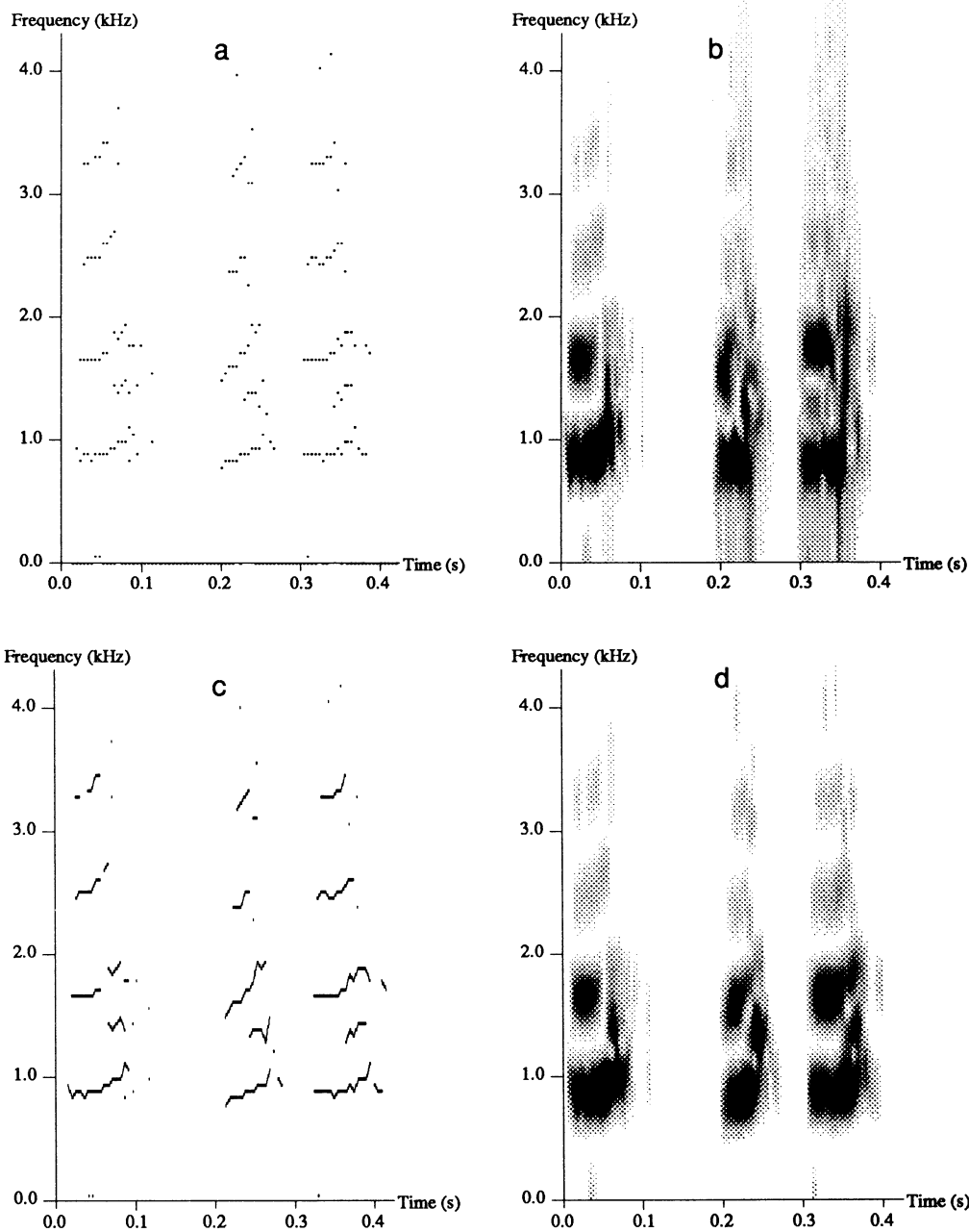


Figure A1.9 The effect of linking on sound reconstruction of elements from the call of the Redbilled Hornbill (*Tockus erythrorhynchus*). **a:** Unlinked peak frames extracted from the call. **b:** The reconstruction from the peak frames in **a**. Note the greater spread of components in the second and third elements, noise effects introduced by abrupt changes. **c:** The linked peaks (linking done by hand using LINKPEAK). **d:** The reconstruction from the linked peak frames in **c**. The result is much closer to the original (see Figure 4.7b in the document body).

A1.2.2.2 LINKPEAK

The previous section, SOUNDGEN, explained the purpose of links between two peaks in different frames. The task of LINKPEAK is to establish these links. When SPECTRA broke the waveform into separate frames, the ideally continuous evolution of the frequency components in the signal was converted to a set of discrete samples, separated in time by the duration of the window. The sequence of frequency components could, however, be reinterpolated to form a continuous function, as SONAGRAM indeed does. That is because each frame has an identical number of components, and thus component N is automatically "linked" to the N'th component in the next frame.

After the PEAKPICK processing, there is no longer a fixed number of components (peaks) in each frame. Moreover, significant peaks change their positions from frame to frame. While a peak may occur at the N'th position at one moment in time, the next moment may see the N'th position's magnitude fall below the significance threshold, and instead the N+1'th position assume the role of significant peak. This is the situation illustrated in trace A of Figure A1.10. Each successive peak in trace A is clearly a "part of the same sound" - the peaks should be interpolated to produce a continuous rising whistle. At the point where trace A gives way to trace B, however, should the last peak in A be linked to the first in B? From the point of view of an automated "link finding" algorithm, there is not much difference between doing so and linking the other peaks in trace A. Trace C shows how a second, simultaneous trace introduces another question - should the trace B peaks be linked to the following peaks in trace B or to the following ones in trace C? Once again, an algorithm needs some criterion by which it can distinguish the linkable peaks in the next frame from the non-linkable ones.

Certainly, a starting point for such a criterion is the distance between two potentially linkable points. In many (if not most) cases, it may be argued that a peak in the next frame that is within some maximum distance (measured on the sonagram trace) from a peak in the current frame, should be linked to it. A theoretical justification for this criterion, an indication of when it breaks down (as it surely does in some cases), and a useful value for the maximum distance are not immediately apparent, however. Together with this lack of practical knowledge is the uncertainty of the importance of the links - perhaps they are not necessary when used in a sound reproduction scheme.

Instead of spending time experimenting with algorithms in an attempt to answer these questions, it was decided to make the linking process a manual one. In this way, the links could be chosen without a strict algorithmic basis. The "correct" choice itself is usually apparent to an observer, who is likely to have access to the original sonagram for comparison. At the same time, the operator may experiment with different hypotheses for linking algorithms, albeit at the expense of increased tedium. LINKPEAK is an interactive graphical program allowing the user to intuitively specify which peaks must be linked. If linking is found to be necessary, and a good strategy for its automation

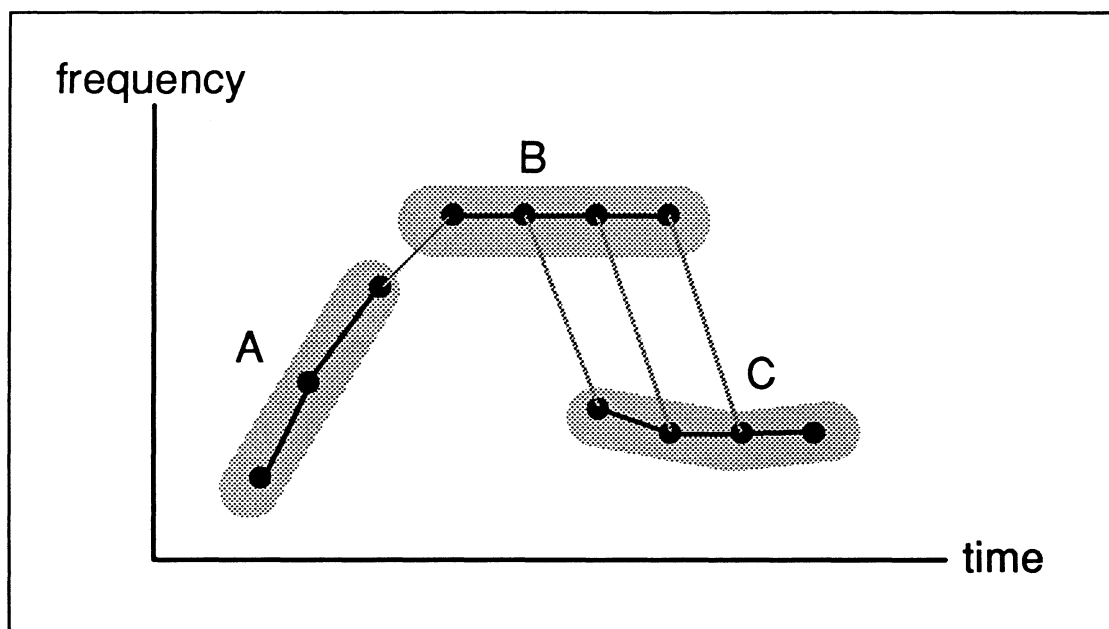


Figure A1.10 Sonagram of a fictitious bird call with three separate component sounds, denoted by traces A-C. The dots superimposed on the traces are the peaks as would be extracted by PEAKPICK. The lines between dots are the links, indicating that the linked peaks need to be interpolated. Solid lines are correct links, light lines are possible errors that an automatic linking algorithm might make.

is found, LINKPEAK can be replaced in total by an appropriate algorithm-based program. Note that LINKPEAK processes potential sound data before that data is stored in a bird call reproduction system,, and would not be part of such a system itself. It is therefore completely separate from the mainstream recognition, experimentation and reproduction processes in the BCR system.

Inputs

LINKPEAK operates on a linked peak frame file in place. It displays the peaks and links that are already set, and allows modification of these links, saving the modified data back in the original file. Thus its main input (and output) is the file of linked peak frames. Its only other input is 'Number_of_frames:' from the Bird Call Descriptor, which provides information on the length of the input file, and the name of the file itself, which is a parameter passed to the program by the user.

Program Structure

LINKPEAK, like DRAWER, is a standalone program for the graphical manipulation of data. It uses the same system of menu-driven pointer-oriented procedures, based on the ICON library of routines. Before describing the specific behaviour of LINKPEAK, a short outline of it's structure

is given, to help those who wish to examine its code. Unlike other BCR programs, it is neither short nor well-structured, and this digression attempts to justify that shortcoming. Please refer to the descriptions of ICON and DRAWER elsewhere in this document, for details not discussed here.

Like DRAWER, LINKPEAK is not intended to be a permanent part of the software, but rather a temporary means of experimentation and visualization. It is a specialist application rather than a public utility, and as such is not designed as a maintainable, reusable or expandable software component. Instead it achieves simplicity and efficiency of use, while attempting to save programming effort by using available building blocks.

Being based on the ICON library, the flow of program control is governed by procedures called from the **Menu** procedure. Briefly, **Menu**'s action is to place a list of options on the screen, and then to call a specified procedure for each option chosen by the user. The text of the option labels is placed in a number of string variables, pointers to which are passed to **Menu** when the latter is called. Together with each label pointer, a pointer to the associated procedure is also passed¹. This means that the main program need only initialise itself and then call a main **Menu** procedure. Other **Menu** procedures are called as submenu options from this main menu. Besides choosing options, the user may exit from a menu by pressing ESC, and a menu option procedure may also specify that the menu be closed on its return by setting a global variable, **MenuQuitFlag**. The latter is defined in ICON and is the only variable in that unit's interface.

Due to a practical limitation of the Pascal language, procedures passed to the **Menu** procedure must have no parameter list². Instead, the "parameters" are global variables. Thus programs based on ICON have a "flat" structure - a set of menu option procedures (all defined at the top level of scope) and a set of global variables (i.e. also defined at the top level). This is not preferred programming practice, being less readable and leaving "private" parameters open to corruption by

¹ In Turbo Pascal (versions 5 and upwards) this is achieved using a formal parameter of type **procedure**. This allows the actual parameter to be the procedure name, rather than an explicitly passing a pointer to that procedure. In any case, the effect is that the address of that procedure is passed, and Turbo Pascal has the means to call such "procedure variable" - i.e. effect a subroutine branch to that address.

² A limitation of the language is that indefinite parameter lists cannot be simply handled - the procedure variable's parameter list format must be known at compile time. This means that in order to pass a variety of procedures to a general procedure dispatcher like **Menu**, some sort of "trick" must be used. In the present case, as explained later in the text, global variables are used as parameters. A more sophisticated solution is to pass a function, which takes another such function as parameter, and returns a value of interest under some conditions, but calls its own function parameter under others. Thus the parameter list becomes a "nest" of function calls, which can be of indeterminate length. However this method is esoteric and probably limits the parameters to a predefined set of types. It was not used in the ICON unit.

procedures which should not have access to them. As mentioned previously, however, these programs are not meant for public consumption, and allowed for rapid development, and efficient operation when used by their author. They are personal tools.

Actions

LINKPEAK begins by entering graphics mode initialising some global variables (genuinely global values, as opposed to pseudo-parameters for menu option procedures). **Xscale** and **Yscale** determine the dimensions of the graph that will soon be displayed, and these are variable for two reasons: **Xscale** is switchable by the user - allowing a zooming action - and both parameters are adjustable (using the **GetMax**- functions) to make the graph reasonably fill the available screen space, which may vary from screen to screen. **StartFrame** is the first frame in the file that will be displayed, used by **DrawLinkedPeaks**. **OldEditX** and **OldEditY** are used to store the screen position where frames are/were editable by the user, used by **EditFrames**.

The length of the bird call is checked. If it will fit entirely in the screen, the "select left edge" option in the main menu is disabled. See the option description below. The input file name passed as command line parameter to the program is checked, and if it does not exist, the user is prompted for a valid name, using ICON's graphical line editing routine, **Prompt**.

If the user being prompted for the file name did not choose to leave the program, the initial screen display depicting the contents of the linked peak file is drawn, and the main (and only) menu is invoked. From here on, the action of the program is determined by the procedures called in response to menu options chosen. When the user is finished with the program, he/she will return to the main menu and press the ESC key. This will cause the **Menu** call to return, and the main program then closes the linked peak file and exits graphics mode before terminating.

The procedure which displays and redisplay the peak frames is **DrawLinkedPeaks**. This routine simply reads the linked peak frame file starting from the frame number specified in **StartFrame** and displays each frame (contained in the working structure **lpf**) as follows. A peak's frequency is translated to a height, by multiplying it by **Yscale**. The frame number (relative to **StartFrame**) is translated to a horizontal displacement from the left edge of the screen, by multiplying it by **Xscale**. A point is plotted at the position thus obtained. So all peaks in a frame will be vertically aligned, at different heights corresponding to their frequency values. When a peak has a non-zero link value, the position in the previous frame (remembered by **oldlpf**) of the corresponding peak is joined to the current peak position by a line. The result is a display similar to Figure A1.11.

DrawLinkedPeaks is called by other procedures whenever the screen needs to be updated - to erase an Edit window, to zoom, or to move the section of the file currently being viewed. The

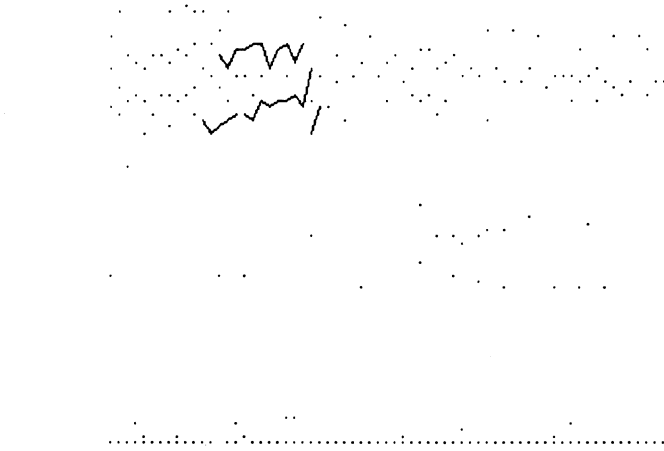


Figure A1.11 Portion of a peak frame display produced by LINKPEAK. The main display remains on the screen, and an edit window containing a shorter, enlarged but similar version appears overlayed in response to selection of the Edit option - see text.

display of the file is thus constantly on-screen. What follows is the option-by-option description of the various functions available through the menu structure, which allow manipulation of this graphical representation of the file.

Main menu: Edit, Expand/Contract, Clear links, L edge, R edge

The menu label strings are defined as global variables.

Edit (procedure EditFrames)

This is the most important procedure, it edits the frames specified by the user. A call to the **Pointed** function obtains the frame number, relative to the first displayed frame, of the frame the user wishes to edit. **Pointed** allows the user to place the graphics cursor (arrow) somewhere on the graph, and calculates the frame number based on that position. **EditFrames** calculates the absolute position of the frame in the linked peak frames file, and reads in a number of frames from that point, equal to the constant **editframeno**. These frames are placed in a data structure (**ef**) that the user will be allowed to edit. Thus **EditFrames** allows editing of a region of the entire call.

The editable frames are now displayed in an independent "window" - a rectangular area of the screen overlayed on the main graph. The area to be used is defined as a local coordinate system using **setviewport**, and a line border is drawn. The editable frames are now displayed at a larger scale inside this window by **ShowEditFrames**. The latter simply reads the **ef** structure and renders its peaks in an identical manner to **DrawLinkedPeaks**, using **editxscale** and **edityscale** instead of **Xscale** and **Yscale**, so that the edit window is offset from the main display by its different (larger) size.

Before editing begins, the user is asked whether to clear existing links within the edit window. If the edited section already contains links, it is likely that only a few of these links will need to be changed, but this option allows all the links to be cleared, which would be a tedious operation if performed one at a time. The links in the entire file can also be cleared from the menu.

The editing loop consists of two modes, that corresponding to the outer Repeat...Until construct and that corresponding to the inner Repeat...Until construct. The outer loop consists of three actions: Update the edit window (to reflect changes made in inner loop), ask user for next link to edit, and enter the inner link editing loop. The first action is achieved by calling **ShowEditFrames**, the second by calling **GetGraphInput**, and the third is explained below. **GetGraphInput** is a general-purpose ICON procedure that allows the user to point the graphics cursor within a specified area. It returns with the pixel coordinates of the cursor when a specified set of keys are pressed. In this instance the pointable area is the entire edit window, and the keys allowed are #1 and #2 (ICON codes for the mouse left and right keys) and #13 and #27, the ESC and ENTER keys. The user positions the cursor near the peak to be edited, and pushes the left mouse key or ENTER to commence editing. If the right mouse key or ESC is pushed, the outer editing loop terminates.

The inner editing loop allows the chosen link to be altered or deleted. Before entering it, the chosen link is determined from the cursor position returned by **GetGraphInput**. The peak frame number is derived from the cursor's horizontal coordinate by dividing by **editxscale**, which has (in effect) units of pixels/frame. The peak chosen within that frame is derived from the cursor's vertical position. **NearestPeak** calculates all the vertical positions of the peaks in a given frame, using **edityscale**, and compares the results with the given y coordinate. The peak with the closest coordinate is chosen. The chosen peak may have a link already. In that case it's value is stored in **oldlink**. If there is no link, a guess is made as to which peak in the previous frame the user would like to link to the chosen peak. This is done by simply choosing the nearest peak in the previous frame using **NearestPeak** once again. This guess is stored in **oldlink**.

Just before the inner editing loop is entered, the link of the chosen peak is flashed off and on, then left on. This is accomplished through a call to **DrawLink**, which takes the peak numbers in the current and previous frame and draws a line between them in a specified colour. The colour used is different to the colour used by **ShowEditFrames**, so the user sees the chosen link highlighted against the other links. The flashing of the link is also for this purpose, to ensure that monochrome screen users will also be informed which link is chosen.

The inner loop also starts with a call to **GetGraphInput**. This time, the cursor is only allowed to move in a vertical path at location of the chosen peak. In addition to the valid keys of the previous mode, #32 (SPACE) is also allowed. **GetGraphInput** is called here in a mode which does not wait for one of the keys to be pressed before returning. It simply returns as soon as the cursor is moved,

passing back the cursor position and keys pressed (if any). With the new cursor position, **NearestPeak** is invoked to find the nearest peak in the previous frame. The current link is erased and a new link to this new peak is drawn (from the current peak). The peak in the current frame that was chosen does not change, so the effect of this loop is to move a link between it and any of the peaks in the previous frame, in response to the cursor's movement. When the user is satisfied that the link drawn is the one desired, he/she presses the left mouse key or the ENTER key. The loop ends, and the **ef** structure is updated with the value of the chosen link (previous frames peak number). When the outer loop iterates, **ShowEditFrames** will show this new link in the "permanent" colour. The loop also ends if the SPACE bar was pressed, in this case the chosen **ef** link is cleared. Finally, if the right mouse key or ESC is pressed, **ef** is left as it was before the inner loop began - no change is made, and the user is back in the outer editing mode.

After the outer editing loop terminates, the frames in **ef** are written over the appropriate frames in the linked peak frame file. **DrawLinkedPeaks** updates the main display, thereby erasing the edit window and letting the newly-written changes be reflected on screen. **EditFrames** terminates.

To summarize the action of **EditFrames** from the user's point of view, let us start at the main menu level. The Edit frames option is chosen. The cursor is moved to the part of the graph to be edited, and the left button (equivalent to ENTER in all cases) is pressed. The edit window appears, containing a larger version of the graph, but only a small number of frames, those in the immediate vicinity of the specified location. The "Clear links" prompt appears and only if the Y key is pressed are the links in the edit window cleared. Any other key (say the left button, since that was the last thing pressed and thus the easiest to "reach") causes the prompt to disappear and enters editing mode 1 (link choice mode). This mode is evident because the cursor can be positioned anywhere in the window. Pressing the right mouse button (equivalent to ESC in all cases) in this mode exits editing and returns to the main menu level, with no changes having been made.

If not wishing to escape, the user now points with the cursor, near the link or unlinked peak to be edited, and presses the left button. Editing mode 2 (link alteration mode) is entered. This mode is evident because the chosen link flashes (or a new link appears where there was none) and changes colour. The cursor is restricted to vertical movement only, and the link jumps from peak to peak in the previous (left-adjacent) frame, in response to the cursor movement. If the cursor was inaccurately placed in mode 1, the "wrong" link may have been chosen. In that case the user presses the right button (synonymous with the escape action throughout the LINKPEAK program), thus re-entering mode 1 with no changes made. If the correct link was chosen, it is now altered by moving the cursor and then pressing the left button, or by pressing the SPACE bar. In the latter case, the chosen link is cleared. After any of these actions, editing mode 1 is re-entered, so that a new link can be chosen for alteration. When all the links in the window have been set as desired,

the right button is pressed, the window disappears and the changes are reflected in the background graph. The main menu level is now current, and to edit a new area of the graph, the Edit option must be chosen.

Expand/Contract (procedure ExpCon)

Allows a "zoomed" view of the main peak frame graph. This is accomplished by modifying **Xscale** and **Yscale** and calling **DrawLinkedPeaks** to redraw the picture. The default value for **Xscale** is 2, meaning that one frame is displayed for every two horizontal coordinates. When Expand is chosen from the menu, **ExpCon** changes this value to 5, so that frames are 5 pixels apart. Similarly, **Yscale** is changed from **getmaxy div 16** to **getmaxy div 8**. The latter value means that peaks with a frequency of 8 kHz will reach the top of the screen, whatever its size. 8 kHz is the highest expected frequency in a bird call. The default value is half of that, so that changing from the default to the expanded view does not overly change the aspect ratio of the graph. Once the expanded view has been chosen, **expandcontractstr**, the variable containing the text of the Expand option, is changed to "Contract". Thus the menu option changes to Contract, and when this option is chosen, the view toggles back to the smaller, default size.

Before drawing the graph, the data is checked to see whether it overflows the screen. The menu options that deal with "moving" the graph left or right to expose different parts of it are altered accordingly. Please see the L Edge and R Edge options below.

Clear links (procedure ClearLinks)

This simple option clears all the links in the linked peak frame file. It does so by copying the frames one by one into a new file, setting each frame's peaks' links to 0 on the way. Then the old file is erased, the new one renamed to the old one's name, and **DrawLinkedPeaks** is called to show the effect of clearing the links.

L Edge and R Edge (procedures LeftEdge and RightEdge)

These two procedures allow the user to pan the graph left or right, exposing parts of it which flowed off the edges of the screen. They always check, after performing their panning actions, whether or not the graph still overflows off the screen. If not, the menu option which would allow further panning is changed to **inactiveconst**, a string of periods ("....."). Choosing this blanked option still calls the appropriate procedure, but checks inside the procedure cause it to have no effect if it cannot pan further.

The way in which the user specifies panning is as follows: The L Edge or R Edge option is chosen. The graphics cursor is then moved to some horizontal position (the vertical position is irrelevant), and the left button pressed. The position thus indicated then becomes the new left or right edge of the screen, as appropriate. So to pull the graph to the left, (i.e. to see what currently flows off the

right edge), L edge is selected and a position somewhere on the graph indicated. This point becomes the new left edge, so the further to the right this point was, the more "new" graph will be pulled in from the right.

The procedures start by calculating the centre coordinate of the screen, for use with the **Pointed** call which follows. The number of the frame which will be at the left edge of the screen when the last frame is at the right edge is calculated and stored in **lastleftedge**. This is useful because the peak frame file is always read "from left to right" - the leftmost frames are encountered first in the file. The procedures now check whether or not their action is possible. For **LeftEdge**, this means checking if **StartFrame**, the frame currently being displayed on the extreme left, is less than **lastleftedge**. If not, the graph is not overflowing of the right of the screen and there is no need to pull it further left. **RightEdge** must check whether **StartFrame** is greater than 0, which if true indicates that there are unseen frames to the left of the screen, and the graph can be pulled further left.

The **Pointed** function is called to get the desired edge position from the user. As before (when used in **EditFrames**), it returns the frame, relative to **StartFrame**, that has been chosen, and evaluates to TRUE if the left button (ENTER) was pressed, and to FALSE if the right button (ESC) was pressed. In the latter case, the procedures terminate, cancelling the panning request. By remembering the old horizontal position in **OldLeftEdge** and **OldRightEdge**, and passing these values to **Pointed** as starting position for the cursor, the menu options "remember" the amount last panned, so that choosing the option repeatedly and pressing the left button without moving the cursor causes a repeated pan of a constant amount - similar to a scroll.

To perform the pan, all that is needed is to change the value **StartFrame** and call **DrawLinkedPeaks**. However it is possible that the pan requested is larger than that required to reach the end or beginning of the file, so this condition is checked for. **LeftEdge** increments **StartFrame** by **relframe**, the value returned by **Pointed**. If the new **StartFrame** is beyond **lastleftedge**, it is clipped to the latter value. For **RightEdge**, **StartFrame** is decremented by the number of frames from the indicated position to the right edge of the screen. This is $\langle \text{frames per screen} \rangle - \text{relframe}$, where $\langle \text{frames per screen} \rangle$ depends on the horizontal pixel resolution of the screen and the current **Xscale** value. If this value is less than 0, **StartFrame** is clipped to 0. Because the procedures work with frame numbers rather than screen positions, relying on **DrawLinkedPeaks** to render the new graph, they work no matter what the current **Xscale** value is - i.e. both in zoomed and unzoomed mode.

As mentioned before, the procedures also check whether they can reasonably be called again and if not, blank out the corresponding menu option. **LeftEdge** must check whether there is any further overflow off the right of the screen ($\text{StartFrame} < \text{lastleftedge}$), and disable "L Edge" if not. In

case the "R Edge" option was blanked by a previous **RightEdge** call, **LeftEdge** must also enable the "R Edge" option if the new left edge is not the first peak frame (**StartFrame** > 0). **RightEdge** must perform the complementary checks and actions.

A1.2.3 The Bird Call Descriptor manipulator: BCDMANIP

The Bird Call Descriptor (BCD) system plays a central rôle in managing and defining the practical aspects of the BCR system. It has multiple functions: It influences the way the software is written, and the format and organization of the files produced. It provides software processes with communication capabilities between themselves and with human operators. It helps enforce conventions so that the proliferation of data is managed in an organized way. It is therefore an important part of a working experimental bird call recognition system.

These aspects of the BCD system are now discussed one by one, both from the point of view of their use in a program, and giving some technical details of how they are implemented.

Organization of software into processes: PutCurrentProcess and CRP.DAT

BCDMANIP provides a mechanism for a program to register itself as a *process*, meaning that it wishes to participate in the BCD system. Registration is done by calling the procedure **PutCurrentProcess**, passing the process name as the parameter. The program can use any name it wishes - it is up to the programmer to ensure that it does not belong to another process, that it is one that is known to the BCD system. For example,

```
PutCurrentProcess ('PEAKPICK');
```

registers the calling program as the PEAKPICK process. The registration should be done once at the start of the program. It is possible to call **PutCurrentProcess** many times, but this makes it less valuable as a means of isolating modules as separate processes. In the BCR system, the pipeline processes all use the name of their module (unit) as the process name, and perform the registration once, before any other action. The current process name is stored in a text file called CRP.DAT in the current directory, rather than in a BCDMANIP global variable. This allows the user to see which process was last active in the event of an unexplained failure, assuming a number of processes were invoked in sequence, as would be possible in the BCR system.

Process to process communication

One of the primary functions of the BCD system is to allow separate programs (registered as processes) to pass data amongst themselves. By being able to do so, the BCR system could be written as a set of partially independent modules, rather than a single huge program. Besides being far easier to manage, understand and maintain, and solving practical memory size problems, the separate process design means that the BCR system is easily adapted to a multitasking, or even parallel processing environment. Some elaboration on this last point is to be found in chapter 7.

Interprocess communication is achieved through access to a common data entity, the Bird Call Descriptor¹, implemented as the text file 'BCD.DAT' in the current directory. One process may write a datum to the file, and another may read it. The data in the file are organized in named fields, much like a Pascal record or C structure. They are also organized in other ways, as will be mentioned later. Like in a record, fields may have different types, and BCDMANIP provides a system of type checking to help avoid programming errors resulting from in type mismatches.

Traditionally, global data in a software system is avoided despite its convenience. If accessible by any part of the software, global data weakens the compartmentalisation of the software, so that unrelated routines have the potential to influence each other without meaning to do so. This makes for fragile, hard-to-maintain code. The BCD system, which makes data global to any number of processes separated physically and in time, attempts to reduce the inherent dangers by implementing a system of access control.

The data organization, type management and access control are all implemented by requiring processes to access BCD.DAT data only through the **BCD** procedure provided by BCDMANIP. Although there is no way to prevent an uncooperative program from reading and writing to BCD.DAT in any way it desires, in practice the convenience offered by **BCD** (which can parse data stored as structured text) means that it is never tempting to do so. All the programmer has to do to ensure compliance with the BCD system is therefore to use **BCD** for any interprocess communication.

The form of a call to **BCD** is as follows:

```
BCD ('read TheField: integer', TheInteger);
```

The first parameter is a command string containing, at the minimum, the access type ('read' or 'write') the name of the desired field (in this case 'TheField:'), and the type of second parameter. It may also include information on where to find the field, as explained later. All field names are expected to end with colons. The second parameter (in this case **TheInteger**) is a variable of any type, which either contains the value to be written to the Bird Call Descriptor, or will receive the value read from it, as in this case.

BCD.DAT is a text file, whose format is explained later. Fields consist of the name followed by the numeric or other characters forming the value. **BCD** searches for the named field in BCD.DAT (producing an error message if this is not found) and then uses the type name passed in the command

¹ The name "Bird Call Descriptor" which is used in various forms throughout the BCD system should not be taken to imply that the system is only applicable to the bird call recognition application. Although designed with the BCR system in mind, it provides quite general services which could well be useful in any large system comprised of separate programs.

string in two ways: The type is used to determine how to convert the text comprising the value, and the type name is compared with the known type of the field. The type of any field in BCD.DAT is recorded along with access information in another text file, BCDACCESS.MAP. In the BCR system, the information in BCDACCESS.MAP applies generally - one does not need a different BCDACCESS.MAP for every bird call, unlike BCD.DAT which describes a specific to a call or group of calls. However in general different BCDACCESS.MAP files might conceivably be required, so the BCD system looks for the file in the current directory rather than in a fixed place.

To clarify the function of the type management feature, let us work through the relevant code section in the **BCD** procedure, for the example given above, reading 'TheField:' into **TheInteger**. We assume that the following line exists in BCD.DAT, assigning 132 to 'TheField:',

```
TheField: 132
```

and that 'TheField' is declared in BCDACCESS.MAP to be of type integer:

```
TheField: INTEGER
```

In the BCDMANIP code, the following section handles the conversion of the string '132' (which has been previously read into **valuestring**):

```

if GetWord (commandline) <> typestring
  then runerror (UserMessage('Type mismatch - note '+'
                               fieldname+' '+typestring));

If dirstring = 'READ' then
begin

  if typestring = 'REAL'
    then val(valuestring, real(value),valerror)
    else if typestring = 'INTEGER'
      then val(valuestring, integer(value),valerror)
      else if typestring = 'WORD'
        then val(valuestring, word(value),valerror)
        else if typestring = 'BYTE'
          then val(valuestring, byte(value),valerror)
          else if typestring = 'LONGINT'
            then val(valuestring, longint(value),valerror)
            else if typestring = 'STRING'
              then string(value) := valuestring
            else runerror (UserMessage(typestring+
                                         ' not a valid BCD type'));

  if valerror <> 0 then
    runerror (UserMessage('The format of the '+'
                           copy(fieldname,1,length(fieldname)-1)+
                           '          value in BCD.DAT'+
                           'does not match its type'));

end;

```

The command string is stored in the **commandline** variable, where subsequent words are retrieved from it using the **GetWord** procedure¹. The first two words, the access direction ('read' or 'write') and the field name have previously been retrieved. The access direction is converted to upper case and stored in the variable **dirstring**. When **GetWord(commandline)** is next performed, as in the code segment shown here, it returns 'INTEGER' - the string declaring the type of the **value** variable (**GetWord** has converted it to uppercase). This is immediately compared to **typestring**, the type of 'TheField:' as declared by BCDACCES.MAP. If the two do not match, the user is trying to read the field into a variable of the wrong type, and an appropriate error message is generated.

¹ The **GetWord** procedure and the **seqrec** record (which is **commandline**'s type) were later combined into the **StringSeq** object defined in GENLIB. Please see GENLIB's descriptions for details on how they are used to parse text files.

The value of **dirstring** now decides whether a string-to-value conversion (usually using the Pascal **val** procedure) or value-to-string conversion (usually using the Pascal **Str** procedure) is required. In this case **dirstring** is 'READ' so the above code segment is executed, otherwise a later one for 'WRITE' would be chosen. The value of **typestring**, in this case 'INTEGER', determines what type the passed-in variable **value** should be treated as (recall that it is an untyped variable parameter and must thus be forced to some type (typecast) when used in an expression). In this case the string '132' is converted as an integer. The value in **typestring** must be one of 'REAL', 'INTEGER', 'WORD', 'BYTE', 'LONGINT' or 'STRING' - some of the basic Turbo Pascal simple types. To give BCDMANIP the capability of handling other types, it necessary only to add to the **if** statement above, and to the complementary 'WRITE' **if** statement. If the passed-in type is unknown, an error message is generated.

Should the string-to-value conversion be unsuccessful, BCDMANIP is able to give a more meaningful error message than if no type information was available - it knows it was unsuccessful because the field was of a format not of the proposed type. The user, knowing the type, can inspect BCD.DAT and compare the value of the particular field with the format that should be.

The type checking in BCD is open to "cheating", in the sense that the user might "lie" or err regarding the type of the variable passed to BCD. For example, if **value** was declared real in the user's code, but the BCD call was "BCD ('read TheField: integer', value);", then BCDMANIP will read the 'TheField:' value into **value**, since it sees that 'TheField:' and **value** have matching types, but it will be read in as an integer. The user's code will interpret it as a real, however, and get an erroneous result. This loophole is all but unavoidable, since the type of a variable cannot be determined dynamically in Pascal. The BCD type checking system is simply an effort to reduce the scope for errors, leaving the above-mentioned one as the only important possibility.

Access via **BCD** is limited to certain processes on a field-by-field basis. Every field in BCD.DAT is declared in BCDACCES.MAP to be accessible by a list of specified processes only. An example of a declaration for a field in BCDACCES.MAP is:

```
TheField: INTEGER
  readers: PROCESS1 PROCESS2
  writers: PROCESS1
```

The declaration states that only PROCESS1 may write the value of 'TheField:', and that both PROCESS1 and PROCESS2 may read it. In the event of an unexpected corruption of a field, examining BCDACCES.MAP will narrow the choice of which processes may be at fault. Altering BCDACCES.MAP is a simple way to eliminate possible sources of error - when a process that is not authorized to do so tries to access a field, a detailed error message occurs, so that the programmer can be sure at all times what access is being made by whom.

Besides logic for checking the type of the field and that the current process (as recorded in CRP.DAT through **PutCurrentProcess**) is in the list of readers or writers of the field, **BCD** also detects various syntax errors in BCDACCES.MAP, again reporting in detail so that these errors are easy to find and eliminate.

Process-operator communication and Data organization

A very important point about the BCD system is that the data is stored in human-readable and human-editable form. In an experimental system it is important that the operator remain in control, and cognizant, of all the data that are generated. If the data are stored in machine-only readable form, more effort is required when inevitable changes need to be made. One requires software to read the old data and convert to the new. While such software is not hard to write, it is still possible to forget to convert some data, and when the problem is discovered its old format may be so obsolete that it is useless and must be discarded. In the BCR system only raw data are in native machine-readable form. All the parameters to the processes, the data which describe the raw data, are stored as text in the Bird Call Descriptor. When a format changes, either only the raw data, or only the parameters are often all that need to change, and this change may be done by hand. At all stages the operator is immediately aware of what is up to date and what is not.

The BCD system also uses the BCD.DAT file to accept input from and even to present output to the operator. Many of the parameters that describe particular tests and runs are entered directly by editing BCD.DAT. The latter file thus acts a permanent record of the key parameters in an experiment - the kind of data that would have to be recorded anyway, but is here *required* by the BCD system. The user is thus forced to keep up to date records.

The BCD system caters for hierarchical organization of data. Fields may be grouped into named blocks. Blocks may be nested within other blocks. **BCD** can be directed to find a particular field within in a particular block, which means that multiple, identically named fields may coexist in BCD.DAT as long as each field is in a different (uniquely named) block.

A block is defined by bracketing a group of fields and/or other blocks with a <name>...'END' pair. The example below shows two blocks nested inside a third, each block having fields of its own. The two nested blocks have identically named fields.

```

OUTER_BLOCK  Start of outer block
  Outerfield1: 103
  Outerfield2: 32.3
  Outerfield3: JSB

  INNER_BLOCK_1
    Innerfield1: 16  Some commentary text
    Innerfield2: -2
  END (INNER_BLOCK_1)

  INNER_BLOCK_2
    Innerfield1: 18
    Innerfield2: -3
  END (INNER_BLOCK_2)
END (OUTER_BLOCK)

```

The indentation of lines in the example is not significant in the BCD system, it is used to enhance readability, as are the bracketed block names beside the 'END' keywords. An 'END' marks the end of the most recently unmatched block, and any text following it (or following any other line such as "Start of outer block" and "Some commentary text") is ignored by the BCDMANIP system.

In order to access the 'Innerfield1' field of INNER_BLOCK_2, the following BCD call can be used:

```

BCD ('read  OUTER_BLOCK  INNER_BLOCK_2  Innerfield:  integer',
MyInt);

```

As the example shows, the field name is preceded by the *path* - the sequence of space-separated block names - required to reach that field. BCD works by scanning the BCD.DAT file for each name in the path in sequence, stopping when the field name (in the command line) is reached (or the end of file is reached). This task is performed by the following code segment, which leaves the file position in the desired block, with **fieldname** containing the (colon-terminated) field name whose value needs to be found.

```

matchedlinenumber := 0;
fieldname := GetWord(commandline);
While not (
    eof (Bird_call_descriptor) or
    (commandline.pos = 1) or
    (fieldname[length(fieldname)] = ':')
) do
begin
    matchfound := FALSE;
    While not (eof(Bird_call_descriptor) or matchfound) do
    begin
        readln(Bird_call_descriptor,BCDline.str);
        BCDline.pos := 1;
        if UpString(GetWord(BCDline)) = fieldname
            then matchfound := TRUE;
        inc (matchedlinenumber);
    End{While};
    if matchfound then fieldname := GetWord(commandline);
End{While};

```

Because of this mechanism it is not always necessary to specify a full path. For example the path 'INNER_BLOCK_2 Innerfield1:' would have worked just as well as the previous example, that is, the 'OUTER_BLOCK' part of the path can be omitted. However, if the path used were simply 'Innerfield1:', BCD would return 16, the value of the *first* 'Innerfield1:' found (which is in INNER_BLOCK_1) rather than 18.

Once BCD is positioned within the correct block, it searches for a field name equal to **fieldname**, and stops when the field is located, or if the keyword 'END' is encountered. The latter case indicates that the desired field was not to be found within the specified block. The former case means that the value of the field is ready to be read, which task is handled by the code described earlier. The rest of the BCDMANIP code is concerned with such things as opening/closing of required files, detecting and reporting syntax or other errors, and in the case of a value being written to the Bird Call Descriptor, a new copy of BCD.DAT being made with the new values in their correctly formatted place, which then overwrites the old copy.

The system used by BCD could be exploited to implement data organization schemes other than a nested hierarchy. For example a linear table could be split into named sections.

The nested hierarchy is used by the BCR system, with data file names being used as block names. A file which is derived from a previous data file, such as a peak frames file which is created by processing a spectral frames file, becomes a block nested within the block of the previous file. A

block, besides containing blocks of the further derived files, contains fields which are the parameters of the process that was used to create the blocks file. Thus the block for A.SF, a spectral frames file, might contain blocks B.LPF and C.LPF for two derived peak frames files, and fields such as 'Window_length:' and 'FFT_length:' which both record (for the human reader) and specify (for the process) parameters used in creating A.SF. Some parameters are derived by processes rather than set by human editing. An example of these are 'Max_magnitude:' in a spectral frame file block and 'PixelsPerKhz:' in a sonagram storage file block.

A1.2.4 Libraries: GENLIB, OBJLIB, TEXTWIN

Throughout the BCR system common utility routines and objects will be found which are not defined in the place they are used - these are the library utilities defined in the GENLIB (General library) and OBJLIB (Object library) modules. The usage and operation (in the context of the BCR system only) of those code segments is explained here.

A1.2.4.1 GENLIB

FileQuery

FileQuery is a function used in many of the processes for testing for the existence of files to prevent unintentional overwriting. A file variable (any file type) is passed to **FileQuery** together with a query type flag value, either **AskOverWrite** or **DontAsk**. The function returns a result which indicates whether the file exists (return value **Exists** for **DontAsk** query flag value) and if the query flag was **AskOverWrite**, the user is asked whether the file may be overwritten and the function returns either **OverWriteOK** or **DontOverWrite** depending on the user's response. If the file does not exist, **DoesNotExist** is returned. If the variable passed in has not yet been associated with a disk file name, the return value is **Unassigned**, and if it is not a file variable at all (any type is allowed since the function deals with all file types), run time error 102 (File not assigned) occurs.

The typical usages in the BCR system are, after assigning a disk file to **filevar** using a Pascal **Assign** statement:

```
If FileQuery (filevar, AskOverWrite) in [OverWriteOK, DoesNotExist]
then rewrite (filevar) else halt (255);
```

And:

```
existboolvar := (FileQuery (filevar, DontAsk) = Exists);
```

The first statement rewrites (recreates or overwrites) **filevar** unless the file already exists and the user typed 'N' (or anything besides 'Y') when prompted by **FileQuery**: 'The file <file's name> already exists. Overwrite (Y/N)?'. Because the creation of **filevar** is typically crucial to the function of the BCR process, the process is aborted (with an exit code of 255 in this case) if rewriting cannot take place. The second statement simply tests whether a file exists (without any user interaction) and stores the result (TRUE or FALSE) in a Boolean variable **existboolvar**.

The function works in the following way: The file variable is cast to a **filerec** type and its fields examined to determine whether it is assigned to a file or not, closed or open, and text file or other file type (typed and untyped files need not be distinguishable). Once its type and status are known the file is opened (or a harmless file operation performed if already open) in the appropriate way,

and the resulting I/O error, if any, then tells whether the disk file exists or if there is some other problem. An I/O result of 0 indicates that the file exists since the operation performed on it was successful, an error of 2 indicates 'File not found', and any other error causes a run-time error (the only other error known to occur is one which indicates that **filevar** was not a file variable at all).

The following BCR modules make use of **FileQuery**: BCDMANIP CLASSIFY ELEMENT LINKPEAK PEAKPICK SONAPS SONAGRAM SOUNDGEN SPECTRA DRAWER.

GraphTransform

GraphTransform is a procedure for transforming an array of N floating-point (type real) numbers to an array of M floating point numbers. The numbers are assumed to have a graphical interpretation, namely as the Y-coordinates of points on a curve whose X-coordinates are placed uniformly along an interval of the X-axis. The input array has N points placed along this interval, and the output array has M such points. The transformation is done by joining adjacent points described by the N input (X,Y) coordinates with straight lines, then recording the Y-coordinates of M points on the resulting curve whose X-coordinates are uniformly placed along the interval. In other words, **Graphtransform** performs a linear interpolation on the input points and returns the desired number of samples of the interpolated curve.

Graphtransform allows a curve specified by a set of discrete samples to be treated as a continuous curve, samples of which may be obtained to any resolution. It is therefore useful when converting between discrete representations of curves, as in the BCR system where a discrete spectrum (obtained from FFT) is displayed (as a songaram line) on a discrete device (a bit-mapped display). The number of pixels desired on screen is not necessarily equal to the number of points produced by the FFT.

The input and output arrays passed to **GraphTransform** may be of any type, to allow efficient and general behaviour. However, the elements of the array are assumed to be of type real and erroneous results will be obtained if they are not. **GraphTransform** must be told the size of the arrays (the number of points they contain, that is, N and M) and if either of these numbers implies an array that is larger than it actually is, a serious error (such as program failure) could result.

Only the SONAGRAM module makes use of **GraphTransform**.

The implementation of **GraphTransform** is straightforward, applying a simple linear interpolation formula between a pair of input points to obtain an output point.

Mean

The **Mean** function, like **GraphTransform**, accepts array variables of any type as input and assumes the elements to be reals. If the passed in array size is wrong, or the elements are not reals, erroneous results will be obtained (although program failure will not result since the array contents are not changed). **Mean** accepts two arrays. It returns the mean (average) value of the numbers in the first array, optionally weighting the mean by the numbers in the second array. The weighting option is turned on or off by the value of a Boolean parameter. **Mean** is used by the **CLASSIFY** process to calculate mean frequency values (such as **freqmean**) and magnitude values (**classification.mag**) in a peak frame. The frequency mean is weighted by the magnitudes, the magnitude mean is unweighted:

```
freqmean := Mean(freqs, PeakCount, mags, TRUE);
classification.mag := Mean(mags, PeakCount, mags, FALSE)
```

Only the **CLASSIFY** module makes use of **Mean**

StdDev

The **StdDev** function returns the (possibly weighted) standard deviation of a set of points. It's parameters are the same as those for **Mean**, and it is used in the same way and with the same provisos.

Only the **CLASSIFY** module makes use of **StdDev**

GenRunError

GenRunError accepts as parameters a number and string. The number is used as the argument to a Turbo Pascal **halt** statement, aborting program execution and returning the number as the DOS exit code, and printing the message on screen. **GenRunError** is used where multiple statements (writing an error message, then calling **halt**) make unnecessarily bulky code. For example it is used often in **FileQuery** 'If' statements as the 'else' clause - thus allowing the existence of a file to be checked for, the user prompted and either a file rewrite or an error message to be produced, all in one concise and readable statement:

```
if FileQuery (Classified_frames, DontAsk) = Exists
then reset (Classified_frames)
else GenRunError (0, 'CLASSIFY couldn't find the Linked peak
frames file');
```

GenRunError also offers the advantage of ensuring that the message appears on screen even the program was aborted while in graphics mode (it enables BIOS text handling by turning Turbo

Pascal's direct video memory access off temporarily). **GenRunError** is not a particular sophisticated run-time error mechanism, and is superseded by the **runerror/UserMessage** mechanism described below.

It is still to be found in certain places in the BCR code, in the following modules: **CLASSIFY ELEMENT SONAGRAM SOUNDGEN SPECTRA DRAWER**.

UserMessage

The **UserMessage** function simply writes the string passed to it (like **GenRunError** direct video access is temporarily turned off so that the text will appear whether the screen is in text or graphics mode) and returns a value of 255. It is intended to be used with a **runerror** routine - the latter is a Turbo Pascal command that generates a run-time error of with specified number. Calling

```
runerror(UserMessage('An error message'));
```

generates a run-time error 255 (general Run Time Error) and prints 'An error message' on the screen, together with the other standard Turbo Pascal run-time error information. This information includes the program address at which the error occurred, and can be used with a debugger to find the place where the error was generated. If the code is run from the Turbo Pascal integrated environment, the position of the error in the source code is automatically found, just as it is for the built-in errors, making the reasons for the error very clear and debugging very straightforward. This is therefore the preferred method for aborting programs due to error conditions.

The modules that make use of the **runerror/UserMessage** mechanism are: **BCDMANIP PEAKPICK SONAPS SYNTAX UPDATE DRAWER** and some **GENLIB** and **OBJLIB** routines.

WriteXY

WriteXY outputs a string to a specified text screen position (X,Y) without changing the current cursor position. If the coordinate specified is 0, the current cursor coordinate is used. Thus **WriteXY** (0,0,'Hello') writes at the same place that write ('Hello') would, but without changing the cursor position, so that the next write statment would overwrite the word 'Hello'. The BCR system uses it to send a number (perhaps reporting on the progress of a calculation) repeatedly to the same place on the screen:

```
WriteXY(70,1,Strung(0,0,frameno)+' ');
```

See later in this section for an explanation of the **Strung** function.

The modules that make use of **WriteXY** are: **PEAKPICK SONAGRAM SPECTRA CLASSIFY ELEMENT** as well as some **GENLIB** routines themselves.

AbsAddr

This function converts a pointer to its linear (24-bit) equivalent address. It is not currently used by BCR modules.

AbsPtr

This function converts a 24-bit address to an equivalent pointer. The pointer is normalised so that the segment part points to the start of a block of memory offset an integer number of 64K blocks from the start of memory. **AbsPtr** is not currently used by BCR modules.

Upstring

This function converts all the letters in the passed in string to upper case. It used by the following modules: BCDMANIP GENLIB SYNTAX UPDATE DRAWER.

Hex

This function converts any integer value into hexadecimal, returning the hex number as a string of characters. It is used by OBJLIB.

UnHex

This function converts a string containing a number in hexadecimal notation to an integer. If the number is larger than the largest integer of type longint, (signed 32-bit integer), the conversion stops before an overflow occurs resulting in only the more significant part of the string being converted. Non-hexadecimal digits are skipped. **UnHex** is not currently used by BCR modules.

Strung

This function returns the text (string) equivalent of the number passed in. The number may be real or integer. Two parameters, **field** and **decplaces** control the conversion. The **decplaces** parameter specifies how many places after the decimal point the converted "number" should have, performing rounding or padding with trailing zero's if necessary. The **field** parameter pads with leading spaces until the string is the specified length, unless the number itself is longer than the field, in which case the full number is returned (it is not clipped to the field length). **Strung** is usually used instead of the Pascal **Str** procedure to enhance the readability of the code, and to avoid intermediate string variables:

```
Report      ('Destroying      element:      '+ClassName[e.class]+'  
' +Strung(5,2,e.durn));
```

The following modules make use of **Strung**: CLASSIFY ELEMENT PEAKPICK SONAGRAM SPECTRA SYNTAX DRAWER.

FitsSpec

This function scans a "filespec" - a string of characters that resolves to a DOS file - in the same way as does DOS. It returns TRUE if the filespec passed in the **Spec** parameter can resolve to the file name (specified by a DOS path) passed in the **Path** parameter. For example,

```
FitsSpec ('\\BAT\\*.bat', '\\BAT\\fruit.bat');
```

returns TRUE, while

```
FitsSpec ('\\BAT\\*.bat', '\\BAT\\fruit.fly');
```

returns FALSE. The characters '*' and '?' have the same meaning as in DOS: '*' can represent any string, '?' represents any character. **FitsSpec** works by splitting both parameters into their directory, name and extension components using Turbo Pascal's **fsplit** routine. The two directory components are compared and must be identical for a TRUE result. The name and extension components are each compared using sub-procedure **WildCardFit**, which expects only the part of the strings preceding a '*' in the spec string to match, and allows a character in the name string to match a corresponding '?' character in the spec string.

Only the UPDATE program (see later) makes use of **FitsSpec**.

ShowFile

This procedure simply displays the text file, the name of which it takes as a parameter. The file is shown one screen at a time, with the user being asked to press a to see the next screen. If there is a problem opening the file, an error message is written to the screen, but the procedure returns without aborting the program.

The UPDATE program (see later) uses **ShowFile** to display a help screen.

DosRead

This calls the built-in DOS line input routine - passing back the string that the user types at the keyboard. It is an alternative to using Pascal's **readln**, which does not provide much in the way of editing facilities. DOS's default line editor does improve slightly upon this situation, providing a buffer that allows the previously entered string to be recalled (F3 key) as well as primitive insertion and deletion characters. However, widely available memory-resident programs can be used to provide a sophisticated replacement for the DOS line editor, and **DosRead** lets a program take advantage of whatever DOS editing system is in use.

No BCR module currently uses **DosRead**.

StringSeq object

This is an object used to parse strings on a word-by-word basis. A string is passed to a *StringSeq*'s **Init** method, and then the *StringSeq* is used rather than the "raw" string. A *StringSeq* treats the string as a sequence of space-separated words, and maintains a current position in the string. Words can be retrieved in sequence by repeatedly calling the **GetWord** method. Once all the words have been retrieved, the **EndOfSequence** instance variable becomes TRUE. At any stage in the sequence, the **Rest** method returns a string containing the remainder of the original string (excluding what has already been read using **GetWord**). The **Reset** method resets the current position so that the next **GetWord** will return the first word in the sequence. The following is an example in which a *StringSeq* object, **seq**, is initialised to the 3-word string 'ONE TWO THREE' and then each word is read and written to screen until none are left. Then **seq** is set to a new string. Note that *StringSeq*.**Init** can be called at any stage - since it does not allocate memory of its own, *StringSeq*.**Done** does not have to be called before a subsequent **Init** (**Done** does nothing and is useful only as a destructor - for disposing dynamically-created *StringSeq*s).

```
VAR seq: StringSeq;
BEGIN
  seq.Init ('ONE TWO THREE');
  while not seq.EndOfSequence do writeln (seq.GetWord);
  seq.Init ('A new string');
END;
```

StringSeq is used by SYNTAX and UPDATE, both of which must parse text files that govern their behaviour. BCDMANIP also does this type of parsing, and should have been converted to use *StringSeq*'s which are based on BCDMANIP's own **GetWord** procedure and *StringSeq* record.

TextLines object

A *TextLines* object is used to enhance access to text files for parsing purposes, just as a *StringSeq* enhances access to strings. A file is associated with a *TextLines* object through the **Open** method (which is a constructor), which takes care of checking that the file can be opened and aborting the program with a run-time error if not. If instead of the filename, the string '%%Reopen%%' is passed to **Open**, the file is assumed to be open, and is simply reset so that its current position is at the start of the file. The **Close** method (a destructor) closes the file, ending the use of the *TextLines* object.

The **FileName** method returns the associated file's name. The **NextLine** method returns a line from the file, as if a **readln** had been used to read into a string (but because it is a function an intermediate string variable for the line is not required). The most useful feature of a *TextLines* object is that it counts the number of lines read, storing this value in the **LinesRead** instance variable. This variable

can be examined directly by the program (**MyTextLines.LinesRead**) although this is not good practice. *TextLines* is intended as a class from which other object types inherit, and these objects can implement methods that use the **LinesRead** value.

In the BCR system, SYNTAX defines a descendant object, *SpecFileTextLines*, that has an additional method, **Expected**. The latter method generates a specified error message, which automatically includes the number of the line (of the file being parsed) that contained the error.

Only the SYNTAX module uses **TextLines**.

A1.2.4.2 OBJLIB

OBJLIB is a set of objects that are general enough to be useful in many situations. Currently OBJLIB contains a base object type *List*, which implements a linear list data structure, and a number of descendant list object types which extend the basic *List* behaviour. The source code clearly explains how to use the objects, so only an outline, some implementation details and some comments on how the objects are used in the BCR system are given here.

List

The basic *List* object contains only three instance variables (fields): **Head**, **Tail** and **Current**, which are all **ListItemPtrs**. A **ListItemPtr** is a pointer to a *ListItem* object. A list is a way of grouping any number (a list may grow and shrink as the a program runs) of these *ListItem* objects in a sequential-access structure. When a *List* is first created (using the constructor method **Init** which takes no arguments), it creates two initial *ListItems* (using heap memory) and makes **Head** and **Tail** point to these items. **Current** is also made to point to the **Head** item for now.

A *ListItem* is an object which is designed to be a part of a *List*. It has a field called **Next** which, being a **ListItemPtr**, can point to another *ListItem*. Initially, the **Next** field of the item pointed to by **Head** is made to point to the item pointed to by **Tail**. We can say that the *head of the list* is connected to the *tail*. As more items are attached to the list, they go "between" the head and tail, so that the head item's **Next** field points to the first 'useful' (additional) list item, and the last useful item's **Next** points to the tail item. All the useful items point to each other in such a way as to form a linear, sequential structure, illustrated in Figure A1.12

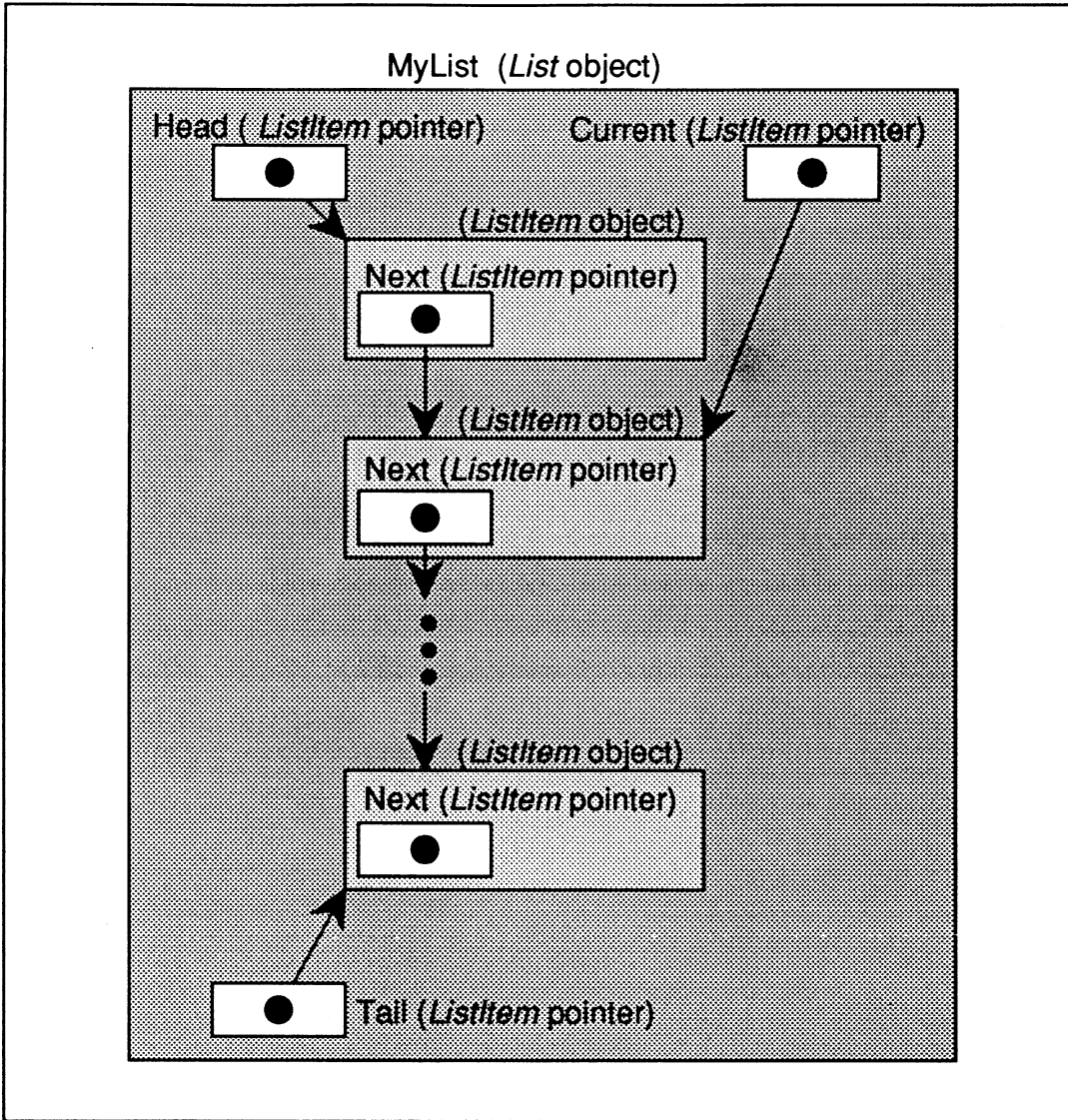


Figure A1.12 A linear linked list as implemented by a **List** object. Boxes denote data structures (objects, variables, fields), the name of the structure appears outside the box, with the type in parentheses. A pointer is a box containing a dot, and an arrow leading from the dot shows the data structure which pointer is pointing to.

An important point to note is that a **ListItem** object has a *virtual* destructor method called **Done**. The **List** destructor (which destroys the list object itself and the two **List**-created items, the head and the tail) is capable of destroying each item in the list by calling that item's **Done** method. **ListItem.Done** is virtual because **ListItem** must be used as a superclass for any items objects that are going to be of any use in a program. The idea is to define a descendant item type, based on

ListItem but with extra fields to contain useful data, a new **Init** method (the old one does nothing) to do custom initialisation of the item, and perhaps a new **Done** to do any cleaning up. Because **Done** is virtual, the new version will be called by *List.Done*, even though the new version wasn't in existence when *List* was coded. The comments in the source code show exactly how to set up a descendant list item system.

The basic feature of a *List* is the ability to add items at will, to any part of the linked chain, using the **Attach** method. **Attach** links in an item at the point where the **Current** fields points to, the *current position* of the list. The **Item** method returns a pointer to the *ListItem* object at the current position. **SetToNext** moves the current position to the next item (moving towards the tail). **SetToHead** moves the current position to the start of the list, the head item. **AtTail** tests whether the current position is at the tail. Note that the head and tail items are always of type *ListItem* (not a descendant type), so they cannot contain useful information. They are "dummy" items, used to mark the start and end of the list and make implementations of various methods more straightforward. If the current position is at the head or tail, **Item** returns a null pointer (NIL). If there are no items besides the head and tail, the list is effectively empty - this condition is reported by the **Empty** method. The methods provided by *List* are oriented towards head-to-tail traversal of the list, as shown in the example below, which writes out the contents of the hypothetical **Namefield** field of all the items in the list **MyList**:

```
MyList.SetToHead;
MyList.SetToNext;
while not MyList.AtTail do
begin
    writeln (Mylist.Item(FALSE)^.Namefield);
    MyList.SetToNext
end;
```

Item can be made to detach the current item from the list, by passing TRUE in the **Detach** parameter. When an item is detached, its neighbours' **Next** fields are updated to point to each other, so that the detached item is no longer accessible through any list items, only through the pointer that **Item** returns. As explained in the source code, it is often desirable, having defined a descendant *ListItem* type, to also define a matching descendant *List* type, so that the new **Item** method can be redefined, returning a pointer to the new item type rather than the generic *ListItemPtr*.

See the BCR SYNTAX module for extensive examples of the use of lists.

Dlist

Dlist is a descendant of *List*, so it can do anything that *List* can do (and *DlistItem* is a descendant of *ListItem*). A *Dlist* (double list) is a more symmetrical structure than list - it can be traversed both from head to tail and from tail to head. It has symmetrical complements for *List* methods: *AtHead* complements *AtTail*, *SetToPrev* complements *SetToHead*, and *SetToTail* complements *SetToHead*. To traverse a *Dlist* in the reverse direction (tail-to-head), one simply uses the loop given in the *List* traversal example above, with all methods replaced by their complements, except that the *SetReverse*(TRUE) method must be called first. The latter sets the direction of the movement of the current position (after attachments and detachments) to be tail-to-head rather than head-to-tail. Because this direction does not necessarily have to be reversed, *Dlist* can exhibit behaviours different from a simple reversal of a *List*, with current position moving in the opposite direction to *SetToNext* or *SetToPrev*.

Dlist is not used by any BCR modules.

Stack

A *Stack* is a *List* with a particular style of access - items are always attached at the head. An Item cannot be accessed without detaching it, and detaching also occurs only at the head. The result is a FIFO (First In First Out) stack behaviour, less recently added items not being available until more recently ones have been detached. Adding to the *top-of-stack*, as the head is called, is termed *pushing*, and is achieved using the *Push* method. Removing from the top-of-stack is called *popping*, and is done with the *Pop* method. The implementation of *Stack* as a *List* descendant is very straightforward, simply adding *Pop* and *Push* methods that always *SetToHead* before *Attaching* or calling *Item* (with *Detach* set to TRUE). Ideally, user access to all other methods should be blocked, but this is not possible in the standard OOP paradigm.

No BCR modules make use of *Stack*.

NumberedList

The *NumberedList* object is a *List* which keeps track of the number of items it has, by incrementing or decrementing its *NumberOfItems* field whenever an item is attached or detached. It is not used in the BCR system.

A1.2.4.3 TEXTWIN

The TEXTWIN module defines a single object, *TextWindow*, that can be used to control the display of text.

TextWindow is an object that manages a rectangular portion of the screen. When initialised using the **Init** method, it is given properties: Size, position, colour (foreground and background for text appearing in the window), whether the window has a border, and a title for the window. **Init** causes the window to appear on the screen and automatically activates it. Once active, subsequent screen output appears in the window, in the appropriate colours. Text is clipped to the window's borders, and can be scrolled inside the window using the standard Turbo Pascal procedures for text manipulation. A window can be deactivated using the **Deactivate** method. When deactivated, the text system (cursor position, colours, etc) reverts to what it was before the window was activated. The window can then be reactivated through the **Activate** method. When destroyed using the **Done** method, the window disappears, leaving behind whatever was on the screen before activation - in other words, the window non-destructively overlays the screen contents. The most typical ways to use a *TextWindow* are to divide up the screen and for pop-up displays.

As screen dividers, a number of *TextWindow* objects are created in such a way as not to overlay each other, and deactivated. Then output is sent during the course of a program to different windows by making the desired window active and then using normal Pascal screen output procedures. After each portion of output, the window is deactivated, so that another window may be used next.

As pop-up windows, a *TextWindow* is created (**Init**) when a temporary message is required, without regard to position on the screen. The message is displayed using **write** statements as normal, and then the window is destroyed (**Done**) before activity in the program resumes from the point where it was interrupted by the pop-up message.

A1.2.5 Miscellaneous utilities: READELEM, READLPF, READSF, UPDATE, SONAPS

Various utility programs that were written when needed are explained here, so that they may be used in future. They are not essential parts of the mainstream BCR software.

READELEM

READELEM is a program that reads a BCR Elements file (.E file) and writes out its contents. The .E file is not a text file, and yet the information in it can be useful to a person formulating a syntax spec. Specifically, when defining element specs for a call, the experimenter would typically gather whatever examples of the call he/she can find. While printouts of these calls could be measured to get the required data, the BCR system can extract it automatically. The calls are run through the system until they have passed through the element discrimination stage. The resulting .E files contain elements as the BCR sees them. Using READELEM to print the files means the experimenter can see or easily deduce the exact boundaries and durations, characteristic frequencies, and class profiles of all the elements detected by the system. The input to READELEM is the .E file name (without extension) and the output is an 'element text file' of the same name, with a '.ETX' extension.

The output file has one element per line, each line has the following format:

```
DURN <duration> FREQ <characteristic frequency> CLASS <dominant  
class> [class profile]
```

where the words in angle brackets '<>' are replaced by numbers, and the [class profile] is of the format:

```
SINGLE <single %> HARMONIC <harmonic %> MULTI <multiple %> NOISE  
<noise %>
```

The need for READELEM would be obviated if the format of .E files were to change to text. If the format changes in any other way, READELEM would have to be changed too, since the file itself does not contain tags identifying the values, it is simply a dump of a sequence of **elementtype** variables.

READLPF, READSF

These two utilities are similar to READELEM in that they output textual forms of BCR system files. The output goes to the standard output device. They do not format the output, as does READELEM, for reading by people. Rather the output is intended as a common exchange format, which can be read by other programs, such as plotting packages.

READLPF takes as input a Linked Peak Frames (.LPF) file, and outputs the frequencies, magnitudes and links of the peaks in each frame, as rows of three numbers. Frames consist of **PeaksPerFrame** such rows, and are separated from one another by blank lines. READLPF does print an explanatory header for the benefit of a person looking at the file. The header contains the file name, the value of PeaksPerFrame, and information enabling the reader to identify the particular frames which are being output. READLPF masquerades as the CLASSIFY process for the purpose of accessing the Bird Call Descriptor.

READSF takes as input a Spectral Frames (.SF) file, and outputs the frequencies and magnitudes of the points in each frame, as rows of two numbers. Frames consist of "FFT_length"/2 (as specified in the Bird Call Descriptor) such rows, and are separated from one another by blank lines. A header containing the file name, the number of points per frame and frame identification information is printed. READSF masquerades as the SPECTRA process for the purpose of accessing the bird call descriptor.

UPDATE

UPDATE is a general utility for maintaining an up-to-date set of data files in face of changing data formats. Although the need for many revisions of the BCR file formats is reduced greatly by the fact that most of the changeable data are stored in the Bird Call Descriptor, the raw information might sometimes still need to undergo revisions. UPDATE allows old files to be converted to new formats using a straightforward, integrated procedure, which is described below.

Whenever a format changes, a special conversion program should be written. The program should be command line driven, accepting the name of the old file as the first (and only) input parameter. The name of this program, and a general mask for all file names that would need to be updated, are then placed in special "update map" file. Also placed in the map file are the dates beyond which each type of file is considered up-to-date. For example, if my old-style data files all have the extension '.ABC', and need to be converted to a new format, I would write a program called ABCUPDAT to do the conversion. ABCUPDAT, when run from the DOS command line, would accept a '.ABC' file and output the new type of file, which could have the same name, but for clarity I give it a new extension, '.DEF'.

Then I would create (or add to) a master map file, perhaps called UPDATE.MAP, inserting the following lines:

```
spec: *.ABC
before: 18/1/92 ABCUPDAT.EXE  Convert .ABC files
                                into .DEF files
                                by adding a XYZ field.
```

Now I simply run UPDATE, passing it the name of the map file, UPDATE.MAP, as a parameter. UPDATE checks all the files that fit the spec "*.ABC" and if it finds any that were created before the 18 of Jan, 1992, the program "ABCUPDAT.EXE" is called to bring it up to date. The text after the "ABCUPDAT.EXE" is commentary text and is ignored by UPDATE.

The "before:" date is necessary for situations in which the old and new files have the same type of name - in the example, if the new files also had .ABC extensions, the date would be the only way to tell which files needed updating. The date is also useful as a record (for the programmer) of when the format was changed. By collecting all the format changes in one map file, all the different file types and multiple subsequent format changes, comprising a large experimental system, can be handled in a single invocation of UPDATE. All the experimenter has to do maintain an up-to-date system is to write the conversion programs as soon as they are needed (while the old format and new format are both fresh in his/her mind) and run UPDATE whenever there is a suspicion that some files are not up to date.

UPDATE can work with a file which is so old that there have been multiple conversions since it was created. In that case, one uses multiple "before" lines in the map file. The order of the lines in the map file should be determined by the dates - UPDATE is not clever enough to do this itself. For example, if .MNO files were first changed on 1/1/80, and then again on 5/5/80, keeping the same names, then it is possible that on the 10/10/80, a directory could contain .MNO files that were created in 1979, as well as brand new .MNO files. To make everything up-to-date by running UPDATE, the map file should contain:

```
spec: *.MNO
before: 1/1/80 MNOFIX1.EXE  Convert Original to New.
before: 5/5/80 MNOFIX2.EXE  Convert New to Latest.
```

Then UPDATE will look for .MNO files created before 1/1/80, and call first MNOFIX1, and then MNOFIX2 on them, bringing them fully up to date. Files that were from before 5/5/80, but after 1/1/80 will have MNOFIX2 only run on them. If the order of the "before:" lines had been reversed, MNOFIX2 would have been called first on all files, and would possibly have failed on the pre-1/1/80 ones unless MNOFIX2 was specifically written to cope with both the very old files and the recently old ones.

To save UPDATE from having to check all files in a directory, a global mask (besides the specific masks in the "spec:" lines in the map file) can be passed to it. For example, if the only updateable files in a directory all begin with "PROJ", one could invoke UPDATE as follows:

```
UPDATE PROJ*.* \SRC\UPDATE.MAP
```

UPDATE.MAP could specify, by using different "spec:" lines, that PROJ*.ABC files be updated differently than PROJ*.DEF or PROJ*.MNO files.

The BCR system currently includes three update programs, PFUPDATE, SSUPDATE and CFUPDATE, for updating old Peak Frame, Sonagram Storage and Classified Frame files. Their listings are given in Appendix B , but they serve as examples only. They are no longer required since all data files are up to date. The map file with which they are used, called BCRUPDAT.MAP, is as follows:

BCR system data object file update information

spec: *.SS

before: 25/10/1989 SSUPDAT.EXE Strips the header - took date
from just before last
updatess.exe creation date.

spec: *.PF

before: 18/10/1991 PFUPDAT.EXE Adds the links (set to zero)
and outputs as a .LPF file.
All .PF files must be updated,
hence the date.

spec: *.CF

before: 17/05/1990 CFUPDAT.EXE Prompts you to run CLASS on
the .LPF file - This redoes the
classification, adding the
average magnitude of the peaks
in the frame. Previously .CFs
had only class and frequency.

SONAPS

This program is a rather rough-and-ready utility for printing sonagrams. Unlike the print facility inside DRAWER, the output is not a screen dump, but converts the Sonagram Storage (.SS) file data directly into PostScript instructions. The conversion algorithm is identical to that in DRAWER (the **DrawSonaPage** procedure). SONAPS was written by translating, statement by statement, the drawing instructions in **DrawSonaPage** and **DrawAxes** into PostScript, defining PostScript procedures to mimic Turbo Pascal and other DRAWER routines when needed. All the screen and user interface code was deleted, as well as the code for handling all the other functions in DRAWER besides the sonagram.

SONAPS requires one to specify an input file (by giving a BCD path) and produces an output PostScript file of the same name as the .SS file, with a .PS extension. In addition, the start and end time (in milliseconds) must be specified, so that the portion of sonagram to be printed is known. Optionally, a title for the graph, which always has both time (with absolute labelling) and frequency axes (the full 8 kHz range), is specifiable, as is the time spacing between time-axis labels. SONAPS is not particularly flexible in its output, especially regarding scaling, but is adequate for technical purposes. It is important to note that scaling and other effects can be achieved by knowledgeable editing of the PostScript file - a **scale** statement may be placed at the very start of the file.

The primary advantage of using SONAPS is the quality of the output - the axis labels are printed in high resolution typeface (some editing of the file can manipulate the size and font of the text if desired), as opposed to the chunky screen-dump quality. SONAPS can also output the full sonagram rather than the screen-sized sections of DRAWER, although it will necessarily overflow onto other pages if the printer cannot handle large paper sizes. Once again judicious scaling will let the full sonagram fill a single page.

Appendix B

Listings of the BCR software

Appendix B Table of Contents

B1 Listings of the BCR software	1
B1.1 BCRTYPES.PAS	2
B1.2 SAMPLE.PAS	3
B1.3 SPECTRA	4
B1.3.1 SPECTRA.PAS	5
B1.3.2 FFTB2.INC	13
B1.4 SONAGRAM.PAS	18
B1.5 PEAKPICK.PAS	21
B1.6 BACKGND.PAS	23
B1.7 CLASSIFY.PAS	24
B1.8 ELEMENT.PAS	26
B1.9 SYNTAX	29
B1.9.1 Simplified SYNTAX.PAS	30
B1.9.2 Full SYNTAX.PAS	44
B1.10 MOUSE.PAS	64
B1.10.1 GETREGS.ASM	68
B1.11 ICON.PAS	69
B1.12 PRTEGA.PAS	78
B1.12.1 GETCBYTE.ASM	92
B1.13 DRAWER.PAS	93
B1.14 SOUNDGEN.PAS	110
B1.15 LINKPEAK.PAS	112
B1.16 BCDMANIP.PAS	119
B1.17 GENLIB.PAS	124
B1.18 OBJLIB.PAS	134
B1.19 TEXTWIN.PAS	140
B1.19.1 SAVERECT.ASM	145
B1.19.2 RESTRECT.ASM	146
B1.20 READELEM.PAS	147
B1.21 READLPF.PAS	148
B1.22 READSF.PAS	149
B1.23 UPDATE.PAS	150
B1.24 PFUPDAT.PAS	153
B1.25 SSUPDATE.PAS	154
B1.26 CFUPDATE.PAS	155
B1.27 SONAPS.PAS	156

B1 Listings of the BCR software

This appendix contains listings of all the source code for the BCR software.
Use in conjunction with the software explanations in Appendix A.

B1.1 BCRTYPES.PAS

```
UNIT BCRTYPES;
{
  Types and constants shared by many modules in the BCR system
}

INTERFACE

CONST MAXFFT = 1024;      {The maximum number of points in an FFT}
CONST MAXWINDOW = 1024;  {Maximum number of samples per time window}
CONST PeaksPerFrame = 5; {the number of peaks in each frame}

TYPE NameStr = string[8];
TYPE peaktype = record
    freq, mag: real;
    link: 0..PeaksPerFrame;
end;
TYPE peakframe = array [1..PeaksPerFrame] of peaktype;
TYPE classtype = (silent, single, harmo, multi, noise, mixed);
TYPE classifiedframe = record
    freq, mag: real;
    class: classtype;
end;
TYPE segmenttype = record
    y: word;
    level: byte;
end;
TYPE elementtype = record
    class: classtype;
    durn: real; {ms duration of element}
    freq: real; {kHz freq of element}
    occurrence: array[single..noise] of real;
end;

CONST
    classname: array [silent..mixed] of string[10] =
        ('SILENT', 'SINGLE', 'HARMONIC', 'MULTIPLE', 'NOISE', 'MIXED');

IMPLEMENTATION
END.
```

B1.2 SAMPLE.PAS

```
UNIT SAMPLE;

INTERFACE
Procedure GetADsamples(Infile, Outfile:string);
{
  Executes the SAMPLE process.
  Converts an incoming signal to an Analogue-to-Digital Samples (.ADS) file.
  The .ADS extension is not assumed for the output file, but should be used
  since it is expected by the BCR system.

  CURRENTLY:
  Convert an input file ('Infile',) of samples sampled by the ILS package to an
  output file ('Outfile') of samples in the format used by the BCR system, by
  removing the ILS header (transferring the sampling frequency contained therein
  to the "Sampling_frequency:" field of the Bird Call Descriptor) and
  converting the data to signed integers.
}

IMPLEMENTATION
USES Crt, BCDMANIP;

Procedure GetADsamples(Infile, Outfile:string);
var
  ILS_samples,
  AD_samples: file of word;
  SampFreq: real;
  ILSword,ADsample: word;
  samplecount:longint;
begin
  PutCurrentProcess ('SAMPLE');

  assign (AD_samples, Outfile);
  assign (ILS_samples,Infile);
  rewrite (AD_samples);
  {$I-} reset (ILS_samples);  {$I+}
  If IOResult <> 0 then
  begin
    writeln ('File handling error: ',Infile);
    halt;
  End{If};

  seek (ILS_samples,61);      {The 62nd word of the ILS header is the s.f.}
  read (ILS_samples,ILSword);
  SampFreq := ILSword/1000;
  BCD('write Sampling_frequency: real',SampFreq);

  samplecount := 0;
  clrscr;
  write('Fetching sample      of ',filesize(ILS_samples)-256);

  seek (ILS_samples,256);      {The first 256 words are the header}
  While not eof(ILS_samples) do begin
    read (ILS_samples, ILSword);
    ADsample := ILSword-2048;
    write (AD_samples, ADsample);
    inc (samplecount);
    If samplecount mod 100 = 0 then begin
      gotoxy (22,1);
      write (samplecount);
    End{If};
  End{While};
  gotoxy (22,1);
  writeln (samplecount);

  close (ILS_samples);
  close (AD_samples);
End{Procedure};

END{UNIT}.
```

B1.3 SPECTRA

The code for spectra consists of two parts - the main BCR process code, in SPECTRA.PAS, and the Borland Turbo Numerical Methods FFT algorithm, in the include file FFTB2.INC. The latter has been very slightly modified from the original, adding an inverse FFT and changing the scaling constant - the changes are to be found at the end of the file.

B1.3.1 SPECTRA.PAS

```
UNIT SPECTRA;
INTERFACE
USES BCRTYPES;

{ $define SConly} {
    Define SConly (by deleting the space preceding the
    '$'character) to compile a program which produces only a
    Single Component file (.SC) is required, not a Spectral
    Frames file as well. This might be desired when a .SF file
    exists but the .SC file, without which peak extraction
    cannot take place, has been misplaced.

    When undefined as above, both a .SF and a .SC file are
    produced.
}

Const
    Last = -1;    {Used when specifying the finishing time with 'GetSpectra'}

Type
    WinUnits = (milliseconds, samples);           {Used by SetWinFft}
    FftUnits = (kilohertz, points);               {Used by SetWinFft}
    TimeWindowType = (Rectangular,                {Used by SetWinFft and Window}
                      Triangular,
                      Hamming);

Procedure GetSpectra (CallName, SpecName: NameStr);
{
    Executes the SPECTRA process.
    Splits a sampled signal (from input .ADS file) into frames and performs a
    Fast Fourier Transform on each frame (output to .SF file). Also produces
    the .SC file needed by the PEAKPICK process.

    'CallName' is the base call name, which is the name of the
    Analogue-to-digital Samples (.ADS) file containing the input bird call.
    'SpecName' is the suffix for the Spectral Frames (.SF) and
    Single Component(.SC) file that are output.
}

IMPLEMENTATION
USES Crt, BCDMANIP, GENLIB;

Type
    Counter = Longint;           {Used for loop counters - for clarity}
    FFTArray = Array [0..MAXFFT] of Real;    {Used for spectral data}

Var
    Error: Byte;                {Code for errors occuring during various operations}

    SamplingFreq: Real;          {The sampling frequency [kHz] for the current signal}
    WINDOWLENGTH: 0..MAXWINDOW; {Number of samples in the time window}
    FFTLENGTH: 0..MAXFFT;        {Number of points used in the FFT}
    NumberOfFrames: longint;      {Number of frames in the signal}
    SpectralFrame: FFTArray;      {Holds the magnitude spectrum of a frame}
    Frame: Counter;              {Holds the number of the frame currently being processed}
    WindowDuration,              {Window duration [ms]}
    FrequencyResolution : Real;   {Frequency resolution [kHz]}
    TimeWindow: TimeWindowType;   {Time window to be used}

    Start, Finish: real;         {desired start and end times [ms], from BCD}
    Windur: real;                {desired window duration [ms], from BCD}
    FreqRes: real;               {desired freq. resolution [kHz], from BCD}
    TimeWindowTypeStr: string[20]; {desired time window type, from BCD}
    Wunits: WinUnits;            {units in which WinDur is expressed, deduced from BCD}
    Funits: FftUnits;           {units in which FreqRes is expressed, deduced from BCD}

Var
    AnySamplesLeft: Boolean;      {Flag signalling end of sample file condition}
    StartTime, FinishTime: Real; {The time at which the first and last samples }
                                {to be processed occur.}
    CurrWindow: String;          {The name of the currently selected window}

Type
    WindowArray = Array [0..MAXWINDOW] of Real;
    WindowArrayPtr = ^WindowArray;
    TNvector = FFTArray;        {REQUIRED BY TURBO NUMERICAL METHODS}
    TNvectorPtr = ^TNvector;    {REQUIRED BY TURBO NUMERICAL METHODS}
```

```

Var
  AD_Samples: File of Integer;           {Samples of the time waveform}
  TimeFrame: WindowArray;                {Single Frame of time samples}
  SinTable,CosTable: TNvectorPtr; {FFT lookup tables used by Turbo Num. Meth.}
  WindowTable: WindowArrayPtr;          {Table of window coefficients - for speed}
  n: counter;                           {Reusable loop counter}
  discard: Integer;                      {Dummy variable}
  NumberOfBits: Byte; {2^N, where N is no. of FFT points - Used by Turbo Num.}

{#####}
{ } Procedure GetNextTimeFrame; { }
{ }
{ Load 'TimeFrame' with the next consecutive set of WINDOWLENGTH samples from }
{ the 'AD_Samples' array. }
{ }
{ This procedure maintains the value of 'Frame', the frame counter }
{ }
{ Reads      : AD_Samples; }
{   FinishTime,SamplingFreq,WINDOWLENGTH : (BCRcomm); }
{ Writes     : TimeFrame, AnySamplesLeft; }
{   Frame : (BCRcomm); }
{ Calls      : none }
{ Called by: GetSpectralFrame : (BCRsig); }
{ }

Var
  NoneLeft : Boolean; {TRUE signifies end of 'AD_Samples' file reached}
  Mediator : Integer; {Used to convert from integer to real value}

Begin
  {GetNextTimeFrame Body}
  {-----}
  If AnySamplesLeft then NoneLeft := FALSE else NoneLeft := TRUE;
  {In case the 'AnySamplesLeft' signal is ignored by the caller}

  For n := 0 to WINDOWLENGTH-1 do

    Begin
      {AD_Samples/(0 if none left) -load-> TimeFrame}
      {-----}
      If Not NoneLeft Then {Dont check Eof unless still open}
      If Eof(AD_Samples)
      Then begin
        NoneLeft := TRUE;
        Close(AD_Samples); {It was opened by 'GetSpectra'}
        AnySamplesLeft := FALSE; {Signal to GetSpectra}
      end;

      If NoneLeft Then TimeFrame[n] := 0
      Else begin
        read(AD_Samples, mediator);
        TimeFrame[n] := mediator;
      end;

      If (Frame*WindowDuration > FinishTime) {The Nth frame starts at}
      Then AnySamplesLeft := FALSE; {time N-1 * WindowDuration...}
      {...So Frame*WindowDuration is the starting time of the Frame+1 th frame}

      {The above check causes an extra frame to be processed if Finishtime is
      not on a frame boundary}

      {-----}
      End; {AD_Samples/(0 if none left) -load-> TimeFrame}

      Inc(Frame); {Maintain the Frame counter}
    {-----}
  End; {GetNextTimeFrame Body}

{#####}
{ } Function Window { }
{ }
{ The formulas (in x) for the window coefficients }
{ }
{ }
{ Parameters } {x: Real; {The independent variable of the formula} }
{ Parameters } Chosenwindow: TimeWindowType) {The particular window chosen }
{ Parameters } : Real; {The window coefficient }
{ }
{ }

```

```

{ The name of the chosen window is placed in the 'CurrWindow' table. }
{ }
{ }
{ Reads      : input parameters }
{ Writes     : CurrWindow }
{ Calls      : none }
{ Called by: SetWinFFT :(BCRsig); }
{ }

```

```

Begin {Window Body}
{-----}
Case ChosenWindow of
  Rectangular: Begin
    Window := 1;
    CurrWindow := 'Rectangular';
    End;

  Triangular: Begin
    If x < (FFTLLENGTH-1)/2 Then Window := 2*x/(FFTLLENGTH-1)
    Else Window := 2*(1-x/(FFTLLENGTH-1));
    Currwindow := 'Triangular (Bartlett)';
    End;

  Hamming: Begin
    Window := 0.54 -0.46 * cos (6.28319/(WINDOWLENGTH-1)*x);
    CurrWindow := 'Hamming';
    End;
End {Case};
{-----}
End; {Window Body}

```

```

{#####}
{ } Procedure ApplyWindow; { }
{ }
{ Apply a time window to the samples in 'TimeFrame' }
{ }
{ }
{ Reads      : WindowTable, TimeFrame; }
{ Writes     : WINDOWLENGTH, Error :(BCRcomm) }
{ Calls      : TimeFrame }
{ Called by: GetSpectralFrame :(BCRsig) }
{ }

```

```

Begin {ApplyWindow Body}
{-----}
For n := 0 to WINDOWLENGTH-1 do

  Begin {TimeFrame -multiply by coeffs-> TimeFrame}
  {-----}
  TimeFrame[n] := WindowTable^[n]*TimeFrame[n];
  {-----}
  End; {TimeFrame -multiply by coeffs-> TimeFrame}

{-----}
End; {ApplyWindow Body}

```

```

{#####}
{ }
{ $I FFTB2.INC Turbo Numerical Methods FFT procedures }
{ }
{ Note that the 1/Sqrt(N) factor has been removed from the forward FFT }
{ algorithm in FFTB2.INC. }
{ }
{ }
{ Reads      : input parameters }
{ Writes     : output parameters }
{ Calls      : none }
{ FFT Called by: PerformFFT :(BCRsig) }
{ MakeSinCosTable Called by: SetWinFFT :(BCRsig) }
{ }

```

```

{#####}
{ } Procedure PerformFFT; { }
{ }
{ Load 'Spectrum' with the 'TimeFrame' samples, pad remaining 'Spectrum' }
{ values and submit as input to the Turbo Numerical Methods FFT. The spectrum }
{ is output in 'Spectrum'. Calculate the magnitude values and load them }

```

```

{ into 'SpectralFrame' }
{ }
{ }
{ Reads      : TimeFrame, SinTable, CosTable, WINDOWLENGTH, FFTLENGTH }
{ Writes     : SpectralFrame }
{ Calls      : FFT :(Turbo Numerical Methods) }
{ Called by: GetSpectralFrame :(BCRsig) }
{ }
}

Const
  PADDINGVALUE = 0;           {Value for padding the end of the time frame}
Var
  Spectrum:      Record
    Real,Imag: TNvectorPtr      {Work space for FFT operations}
  End;

Begin
  {PerformFFT Body}
  {-----}
  New(Spectrum.Real);           {Create the work space on the heap}
  New(Spectrum.Imag);
  For n := 0 to WINDOWLENGTH-1 do
  Begin
    {TimeFrame/(0s for Imag part) -load-> Spectrum}
    {-----}
    Spectrum.Real^[n] := TimeFrame[n];
    Spectrum.Imag^[n] := 0;
    {-----}
  End;
  {TimeFrame/(0s for Imag part) -load-> Spectrum}
  For n := WINDOWLENGTH to FFTLENGTH-1 do
  Begin
    {PADDINGVALUES -load-> Spectrum}
    {-----}
    Spectrum.Real^[n] := PADDINGVALUE;
    Spectrum.Imag^[n] := 0;
    {-----}
  End;
  {PADDINGVALUES -load-> Spectrum}
  {
  slightly faster way - replace the above loops with:
    move (TimeFrame, Spectrum.Real, WINDOWLENGTH*sizeof(TimeFrame^[1]));
  and dont destroy Spectrum after each usage. Initialize with the following:
    fillchar (Spectrum.Real^, FFTLENGTH,PADDINGVALUE)
    fillchar (Spectrum.Imag^,FFTLENGTH, 0);
  }
  {-----}
  {Spectrum -FFT-> Spectrum}
  {-----}
  FFT_TN (NumberOfBits,FFTLENGTH,FALSE,Spectrum.Real,Spectrum.Imag,
    SinTable, CosTable);
  {-----}
  {Spectrum -FFT-> Spectrum}
  {-----}
  {Spectrum -magnitude-> SpectralFrame}
  {-----}
  For n := 0 to FFTLENGTH-1 do
  SpectralFrame[n] := Sqrt
    (Sqr(Spectrum.Real^[n]) + Sqr(Spectrum.Imag^[n]));
  {-----}
  {Spectrum -magnitude-> SpectralFrame}
  {-----}
  Dispose(Spectrum.Imag);      {Discard FFT workspace}
  Dispose(Spectrum.Real);
  {-----}
End;
{PerformFFT Body}

{
{ To obtain the very first sample, StartTime should be 0. To obtain the last }
{ 'Finish' should be "Last", defined in BCRcomm to be -1. }
{ If StartTime is beyond the end of the file, 'AnySamplesLeft' will become }
{ FALSE. If Finish is beyond the end of the file, the effect is as if it was }
{ "Last", That is 'FinishTime' is set to the time at which the last sample of }
{ the file occurs. 'FinishTime' is used by 'GetNextTimeFrame'. }
{ }
{ }
{ Reads      : input parameters, AnySamplesLeft; }
{ Writes     : SamplingFreq, StartTime, FinishTime :(BCRcomm); }
{ Calls      : none }
{ Called by: User; }
}

```

```

Procedure SetTimes (Start, Finish: real);
Begin
    {InputSignal_is Body}
    {-----}

    {-----}
End;
    {InputSignal_is Body}

#####
{
    } Procedure SetWinFft {
    }
    { Allows supervised setting of the constants FFTLENGTH and WINDOWLENGTH,
    { either directly or by specifying the the window duration (in ms) and the
    { frequency resolution (in kHz). The actual resulting frequency resolution
    { will be higher than specified if it is necessary to adjust FFTLENGTH up to
    { a power of two.
    { Allows choice of a time window via the 'TimeWindow' parameter.
    { The tables used by the FFT and window procedures are created here, so
    { 'SetWinFFT' must be used by the User before 'GetSpectralFrame'.
    { The frame counter, 'Frame' gets its initial value here, so that a call to
    { 'SetWinFFT' restarts processing.
    { 'SamplingFreq' and 'StartTime' must have values before SetWinFft is called.
    }
    {
    { (WinDur: Real; {Time window duration or WINDOWLENGTH value
    { Wunits: WinUnits; {Units of WinDur
    { Parameters } FreqRes: Real; {Frequency resolution or FFTLENGTH value
    { Funits: FftUnits; {Units of FreqRes
    { TimeWindow: TimeWindowType); {The chosen window
    }
    }
    { If Wunits is 'milliseconds' then WINDOWLENGTH is calculated. If Wunits is
    { 'samples' 'WinDur's value is assigned to WINDOWLENGTH. The actual window
    { duration, in ms, is sent to 'WindowDuration'.
    { If Funits is 'kiloherz' then FFTLENGTH is calculated as the power of 2 that
    { gives the closest resolution to 'FreqRes'. If Funits is 'points', FFTLENGTH
    { is given the value of the power of 2 nearest 'FreqRes'. The actual reso-
    { lution, in kHz, is sent to 'FrequencyResolution'.
    { A sampling frequency in kHz must be supplied in 'SamplingFreq'.
    { If the parameters are badly specified, 'Error' will contain a non-zero code
    { and only the calculations of WINDOWLENGTH and FFTLENGTH are performed; no
    { tables are created.
    }
    {
    { Reads : input parameters;
    { SamplingFreq,StartTime :(BCRcomm);
    { Writes : WindowDuration, FrequencyResolution;
    { Error,WINDOWLENGTH, FFTLENGTH :(BCRcomm);
    { Calls : none
    { Called by: User;
    }
    }

Var FftTemp: Real; {Working variables}
    m : Word;
    n: Word;

Begin
    {SetWinFft Body}
    {-----}
    Error := 0; {Error=0 - No error}

    If Funits = kiloherz {Calculate nominal value for FFTLENGTH}
    Then FftTemp := SamplingFreq / FreqRes
    Else FftTemp := FreqRes;

    If FftTemp > MAXFFT
    Then Error := 1 {Error=1 - Too many points/Too few Hz specified}

    Else Begin {Adjust the FFT length upwards to a power of 2}
        {-----}
        n := 1;
        m := 2; {FFTLENGTH will always be >= 2}
        While trunc(FftTemp) > m Do begin m := m shl 1; inc(n); end;
        NumberOfBits := n;
        FFTLENGTH := m; {FFTLENGTH will always be 2^N}
        {-----}
    End; {Adjust the FFT length upwards to a power of 2}

    If Wunits = milliseconds
    Then WINDOWLENGTH := Trunc (WinDur * SamplingFreq)
    Else WINDOWLENGTH := Trunc (WinDur);

    If (Error = 0) and (WINDOWLENGTH > MAXWINDOW)
    Then Error := 2; {Error=2 - Too many samples/milliseconds specified}

```

```

If (Error = 0) and (WINDOWLENGTH > FFTLENGTH)
  Then Error := 3;      {Error=3 - Window too big - need to downsample}

If (Error = 0) and (WINDOWLENGTH < 1)
  Then Error := 4;      {Error=4 - Window too small - need at least one point}

FrequencyResolution := SamplingFreq / FFTLENGTH;    {Record the parameters}
WindowDuration := WINDOWLENGTH / SamplingFreq;

Frame := Trunc(StartTime/WindowDuration)+1;    {Initialise the frame counter}
      { ... Frame=1 at time 0 etc}

If SinTable <> NIL Then Dispose(SinTable);
If CosTable <> NIL Then Dispose(CosTable);
If WindowTable <> NIL Then Dispose(WindowTable);
SinTable := NIL; CosTable := NIL; WindowTable := NIL;

If Error = 0
  Then Begin
    {Create FFT and Window coeff. tables}
    {-----}
    New (SinTable);
    New (CosTable);
    MakeSinCosTable(FFTLENGTH,SinTable,CosTable); {TURBO NUM METH ROUTINE}
    New(WindowTable);
    For n := 0 to WINDOWLENGTH-1 do WindowTable^[n] :=
      Window(n,TimeWindow);
    {-----} End;
  {Create
FFT and Window coeff. tables}

{-----}
End;                                {SetWinFft Body}

#####
{ } Procedure GetSpectralFrame; { }
{ }
{ Call procedures to get a magnitude spectrum of FFTLENGTH samples in
the 'SpectralFrame' array }
{ }
{ Reads      : none }
{ Writes     : none }
{ Calls      : GetNextTimeFrame, ApplyWindow, PerformFFT :(BCRsig) }
{ Called by: User }
{ }

Begin                                {GetSpectralSamples Body}
{-----}
GetNextTimeFrame;      {AD_Samples -loadFFTLENGTHsamples-> TimeFrame}
ApplyWindow;           {TimeFrame -window-> TimeFrame}
PerformFFT;            {TimeFrame -FFT-> SpectralFrame}
{-----}
End;                                {GetSpectralSamples Body}

Procedure GetSpectra;
var
  Spectral_frames: file;
  Single_component: file of FFTArray; {contains the spectrum of a single}
                                         {component at 0Hz, magnitude 1}
  MaxComp: real;                      {The maximum magnitude in the single}
                                         { component spectrum}
  MaxMag, newmaxmag: real;             {used for calculating the maximum}
                                         { component magnitude in the signal}

{}Procedure GetMaxMag (var max: real);
{}
{} [find the largest (magnitude) component in the spectral frame]
{}
{}var i: 0 .. MAXFFT div 2;
{}begin
{}  max := 0;
{}  for i := 0 to FftLength div 2 do
{}    if SpectralFrame[i] > max then max := SpectralFrame[i];
{}End{procedure};

var
  sfilepath: string; {Caches the BCD path to the spectral frames parameters}

```

```

begin (GetSpectra)
  PutCurrentProcess ('SPECTRA');
  sfilepath := CallName+'.ADS '+CallName+SpecName+'.SF';
  BCD('read '+CallName+'.ADS Sampling_frequency: real',SamplingFreq);

{
  The following section replaces a set of input parameters, so some of
  the things are slightly contrived - they were added on to previous
  structure rather than completely renewed. The desired values are read
  from various BCD fields, then checked for consistencies and changed
  if possible, then written back to BCD for use or information.
}

  Windur := -1;
  FreqRes := -1;

  BCD ('read '+sfilepath+ ' Start_time: real',Start);
  BCD ('read '+sfilepath+ ' Finish_time: real',Finish);
  BCD ('read '+sfilepath+ ' Window_duration: real',WinDur);
  BCD ('read '+sfilepath+ ' Frequency_resolution: real',FreqRes);
  BCD ('read '+sfilepath+ ' Window_type: string',TimeWindowTypeStr);

  TimeWindow := Rectangular;
  TimeWindowTypeStr := UpString(TimeWindowTypeStr);
  If TimeWindowTypeStr = 'HAMMING' then TimeWindow := Hamming;
  If TimeWindowTypeStr = 'TRIANGULAR' then TimeWindow := Triangular;
  If (TimeWindowTypeStr <> 'RECTANGULAR') and (TimeWindow = Rectangular)
    then runerror(UserMessage
      ('SPECTRA: Invalid window type specified: '+TimeWindowTypeStr));

  If WinDur > 0
    then Wunits := milliseconds
    else begin
      BCD ('read '+sfilepath+ ' Window_length: word',WINDOWLENGTH);
      WinDur := WINDOWLENGTH;
      Wunits := samples;
    end;

  If FreqRes > 0
    then Funits := kilohertz
    else begin
      BCD ('read '+sfilepath+ ' FFT_length: word',FFMLENGTH);
      FreqRes := FFMLENGTH;
      Funits := points;
    end;

  StartTime := Start;

  assign (AD_Samples, CallName+'.ADS');
  if FileQuery (AD_Samples, DontAsk) = Exists
    then reset (AD_Samples)
    else runerror(UserMessage (CallName+'.ADS'+ ' not found by SAMPLE'));

  seek (AD_Samples, Trunc(StartTime*SamplingFreq));
  if (Finish = Last) or
    ((Finish*SamplingFreq) > FileSize(AD_Samples)) or
    (Finish < Start)
    then FinishTime := FileSize(AD_Samples)/SamplingFreq
    else FinishTime := Finish;

  starttime := 0;
  SetWinFft (WinDur, Wunits, FreqRes, Funits, TimeWindow);
  {Note - this sets or changes WINDOWLENGTH, FFMLENGTH, WindowDuration}
  {and FrequencyResolution}
  If Error <> 0 then begin
    Case Error of
      4: writeln ('Window needs at least 1 point');
      3: writeln ('Window has more points than FFT - downsample or respecify');
      2: writeln ('Number of window samples or milliseconds specified exceeds '+
        'maximum allowed');
      1: writeln ('No. of FFT points or frequency resolution (Hz) specified '+
        'exceeds allowed limits');
    End(Case);
    runerror(UserMessage('Please respecify SPECTRA parameter(s)'));
  End(If);

  BCD('write '+sfilepath+ ' Window_length: word',WINDOWLENGTH);
  BCD('write '+sfilepath+ ' FFT_length: word',FFMLENGTH);
  BCD('write '+sfilepath+ ' Window_duration: real', WindowDuration);
  BCD('write '+sfilepath+ ' Frequency_resolution: real', FrequencyResolution);

  NumberOfFrames := round((FinishTime-StartTime)/WindowDuration);
  BCD('write '+sfilepath+ ' Number_of_frames: longint',NumberOfFrames);

```

```

{$ifndef SConly}
  writeln ('FFTING ',NumberOfFrames, ' frames ');
  assign (Spectral_frames, CallName+SpecName+'.SF');
  if FileQuery (Spectral_frames, AskOverWrite) in {OverWriteOk, DoesNotExist}
    then rewrite (Spectral_frames, (FFTLENGTH div 2)*sizeof(SpectralFrame[1]))
    else halt(0);
  {Store only the positive-frequency half of the spectrum, since the negative
   half (from FFTLENGTH/2 to FFTLENGTH) is symmetrical for real signals}

  maxmag := 0;
  AnySamplesLeft := TRUE;
  Repeat
    writexy (70,1, Strung (0,0,Frame));
    GetNextTimeFrame;
    ApplyWindow;
    PerformFFT;
    blockwrite (Spectral_frames,SpectralFrame,1);

    GetMaxMag (newmaxmag);
    if newmaxmag > maxmag then maxmag := newmaxmag;
  Until not AnySamplesLeft;

  BCD('write '+sfpath+' Max_magnitude: real', maxmag);
{$endif}

{Generate the magnitude spectrum of a single component.
 Use a sine wave with a frequency of half the sampling frequency
 so that the component is exactly in the centre of the SpectralFrame array.
 Normalise the spectrum so that maximum point (the FFTLENGTH/2+1'th point)
 has a magnitude of 1. Save the spectrum in SC.DAT}

for n := 0 to WINDOWLENGTH-1 do TimeFrame[n] := sin (n*3.141593);
{note: using the built-in PI constant here gives strange "noise" in the
 spectrum}

ApplyWindow;
PerformFFT;
MaxComp := SpectralFrame[FFTLENGTH div 2];
for n := 0 to FFTLENGTH-1 do SpectralFrame[n] := SpectralFrame[n]/MaxComp;

assign (Single_component, CallName+SpecName+'.SC');
rewrite (Single_component);
write (Single_component, SpectralFrame);
close (Single_component);

{$ifndef SConly}
  close (Spectral_frames);
{$endif}
dispose (SinTable);
dispose (CosTable);
End{Procedure GetSpectra};

BEGIN{INITIALISATION}
  WINDOWLENGTH := 0; FFTLENGTH := 0;
  SinTable := NIL; CosTable := NIL; WindowTable := NIL;
END.

```

B1.3.2 FFTB2.INC

```
{
  This fast fourier transform algorithm is used by the SPECTRA process. It
  has been very slightly modified from the original, to perform both a forward
  and inverse FFT, with the output being scaled differently to the original.
  The modifications are made near the end of the file and are marked by
  comments beginning "Mod: ".
}
```

```
{-----}
{-                                     -}
{-                               FFTB2.INC                               -}
{-                                     -}
{-   Turbo Pascal Numerical Methods Toolbox                           -}
{-   (C) Copyright 1986 Borland International.                         -}
{-                                     -}
{-   Version Date: 26 January 1987                                     -}
{-                                     -}
{-----}
```

```
procedure TestInput(NumPoints    : integer;
                    var NumberOfBits : byte;
                    var Error      : byte);
```

```
{-----}
{- Input: NumPoints                                     -}
{- Output: NumberOfBits, Error                         -}
{-                                     -}
{- This procedure checks the input.  If the number of points -}
{- (NumPoints) is less than two or is not a multiple of two -}
{- then an error is returned.  NumberOfBits is the number of -}
{- bits necessary to represent NumPoints in binary (e.g. if -}
{- NumPoints = 16, NumberOfBits = 4).                     -}
{-----}
```

```
type
  ShortArray = array[1..13] of integer;
```

```
var
  Term : integer;
```

```
const
  PowersOfTwo : ShortArray = (2, 4, 8, 16, 32, 64, 128, 256,
                              512, 1024, 2048, 4096, 8192);
```

```
begin
  Error := 2; { Assume NumPoints not a power of two }
  if NumPoints < 2 then
    Error := 1; { NumPoints < 2 }
  Term := 1;
  while (Term <= 13) and (Error = 2) do
    begin
      if NumPoints = PowersOfTwo[Term] then
        begin
          NumberOfBits := Term;
          Error := 0; { NumPoints is a power of two }
        end;
      Term := Succ(Term);
    end;
end; { procedure TestInput }
```

```
procedure MakeSinCosTable(NumPoints : integer;
                          var SinTable : TNvectorPtr;
                          var CosTable : TNvectorPtr);
```

```
{-----}
{- Input: NumPoints                                     -}
{- Output: SinTable, CosTable                         -}
{-                                     -}
{- This procedure fills in a table with sin and cosine -}
{- values.  It is faster to pull data out of this -}
{- table than it is to calculate the sines and cosines. -}
{-----}
```

```
var
  RealFactor, ImagFactor : real;
  Term : integer;
  TermMinus1 : integer;
  UpperLimit : integer;
```

```

begin
  RealFactor := Cos(2 * Pi / NumPoints);
  ImagFactor := -Sqrt(1 - Sqr(RealFactor));
  CosTable^0 := 1;
  SinTable^0 := 0;
  CosTable^1 := RealFactor;
  SinTable^1 := ImagFactor;
  UpperLimit := NumPoints shr 1 - 1;
  for Term := 2 to UpperLimit do
    begin
      TermMinus1 := Term - 1;
      CosTable^Term := CosTable^TermMinus1 * RealFactor -
        SinTable^TermMinus1 * ImagFactor;
      SinTable^Term := CosTable^TermMinus1 * ImagFactor +
        SinTable^TermMinus1 * RealFactor;
    end;
  end; { procedure MakeSinCosTable }

procedure FFT_TN(NumberOfBits : byte;
  NumPoints : integer;
  Inverse : boolean;
  var XReal : TNvectorPtr;
  var XImag : TNvectorPtr;
  var SinTable : TNvectorPtr;
  var CosTable : TNvectorPtr);

{-----}
{- Input: NumberOfBits, NumPoints, Inverse, XReal, -}
{- XImag, SinTable, CosTable -}
{- Output: XReal, XImag -}
{- -}
{- This procedure implements the actual fast Fourier -}
{- transform routine. The vector X, which must be -}
{- entered in bit-inverted order, is transformed in -}
{- place. The transformation uses the Cooley-Tukey -}
{- algorithm. -}
{-----}

const
  RootTwoOverTwo = 0.707106781186548;

var
  Term : byte;
  CellSeparation : integer;
  NumberOfCells : integer;
  NumElementsInCell : integer;
  NumElInCellLess1 : integer;
  NumElInCellSHR1 : integer;
  NumElInCellSHR2 : integer;
  CosTerm, SumTerm, DifTerm : real;
  Element : integer;
  CellElements : integer;
  ElementInNextCell : integer;
  Index : integer;
  RealDummy, ImagDummy : real;
  Dummy1, Dummy2 : real;

procedure BitInvert(NumberOfBits : byte;
  NumPoints : integer;
  var XReal : TNvectorPtr;
  var XImag : TNvectorPtr);

{-----}
{- Input: NumberOfBits, NumPoints -}
{- Output: XReal, XImag -}
{- -}
{- This procedure bit inverts the order of data in the -}
{- vector X. Bit inversion reverses the order of the -}
{- binary representation of the indices; thus 2 indices -}
{- will be switched. For example, if there are 16 points, -}
{- Index 7 (binary 0111) would be switched with Index 14 -}
{- (binary 1110). It is necessary to bit invert the order -}
{- of the data so that the transformation comes out in the -}
{- correct order. -}
{-----}

var
  Term : integer;
  Invert : integer;
  Hold : real;
  NumPointsDiv2, K : integer;

```

```

begin
  NumPointsDiv2 := NumPoints shr 1;
  Invert := 0;
  for Term := 0 to NumPoints - 2 do
    begin
      if Term < Invert then ( Switch these two indices )
      begin
        Hold := XReal^[Invert];
        XReal^[Invert] := XReal^[Term];
        XReal^[Term] := Hold;
        Hold := XImag^[Invert];
        XImag^[Invert] := XImag^[Term];
        XImag^[Term] := Hold;
      end;
      K := NumPointsDiv2;
      while K <= Invert do
        begin
          Invert := Invert - K;
          K := K shr 1;
        end;
      Invert := Invert + K;
    end;
  end; ( procedure BitInvert )

begin ( procedure FFT )
  ( The data must be entered in bit inverted order )
  ( for the transform to come out in proper order )
  BitInvert(NumberOfBits, NumPoints, XReal, XImag);

  if Inverse then
    ( Conjugate the input )
    for Element := 0 to NumPoints - 1 do
      XImag^[Element] := -XImag^[Element];

  NumberOfCells := NumPoints;
  CellSeparation := 1;
  for Term := 1 to NumberOfBits do
    begin
      ( NumberOfCells halves; equals 2^(NumberOfBits - Term) )
      NumberOfCells := NumberOfCells shr 1;
      ( NumElementsInCell doubles; equals 2^(Term-1) )
      NumElementsInCell := CellSeparation;
      ( CellSeparation doubles; equals 2^Term )
      CellSeparation := CellSeparation shl 1;
      NumElInCellLess1 := NumElementsInCell - 1;
      NumElInCellSHR1 := NumElementsInCell shr 1;
      NumElInCellSHR2 := NumElInCellSHR1 shr 1;

      ( Special case: RootOfUnity = EXP(-i 0) )
      Element := 0;
      while Element < NumPoints do
        begin
          ( Combine the X[Element] with the element in )
          ( the identical location in the next cell )
          ElementInNextCell := Element + NumElementsInCell;
          RealDummy := XReal^[ElementInNextCell];
          ImagDummy := XImag^[ElementInNextCell];
          XReal^[ElementInNextCell] := XReal^[Element] - RealDummy;
          XImag^[ElementInNextCell] := XImag^[Element] - ImagDummy;
          XReal^[Element] := XReal^[Element] + RealDummy;
          XImag^[Element] := XImag^[Element] + ImagDummy;
          Element := Element + CellSeparation;
        end;

      for CellElements := 1 to NumElInCellSHR2 - 1 do
        begin
          Index := CellElements * NumberOfCells;
          CosTerm := CosTable^[Index];
          SinTerm := SinTable^[Index];
          DifTerm := SinTable^[Index] - CosTerm;
          Element := CellElements;

          while Element < NumPoints do
            begin
              ( Combine the X[Element] with the element in )
              ( the identical location in the next cell )
              ElementInNextCell := Element + NumElementsInCell;
              Dummy1 := (XReal^[ElementInNextCell] +
                XImag^[ElementInNextCell]) * CosTerm;
              Dummy2 := XReal^[ElementInNextCell] * DifTerm;
              RealDummy := Dummy1 - XImag^[ElementInNextCell] * SinTerm;
              ImagDummy := Dummy1 + Dummy2;
              XReal^[ElementInNextCell] := XReal^[Element] - RealDummy;
              XImag^[ElementInNextCell] := XImag^[Element] - ImagDummy;
              XReal^[Element] := XReal^[Element] + RealDummy;
              XImag^[Element] := XImag^[Element] + ImagDummy;
            end;
          Element := Element + CellSeparation;
        end;
      end;
    end;
  end;

```

```

    Element := Element + CellSeparation;
end;
end; { for }

{ Special case: RootOfUnity = EXP(-i PI/4) }
if Term > 2 then
begin
    Element := NumElInCellSHR2;
    while Element < NumPoints do
    begin
        { Combine the X[Element] with the element in }
        { the identical location in the next cell }
        ElementInNextCell := Element + NumElementsInCell;
        RealDummy := RootTwoOverTwo * (XReal^[ElementInNextCell] +
                                         XImag^[ElementInNextCell]);
        ImagDummy := RootTwoOverTwo * (XImag^[ElementInNextCell] -
                                         XReal^[ElementInNextCell]);
        XReal^[ElementInNextCell] := XReal^[Element] - RealDummy;
        XImag^[ElementInNextCell] := XImag^[Element] - ImagDummy;
        XReal^[Element] := XReal^[Element] + RealDummy;
        XImag^[Element] := XImag^[Element] + ImagDummy;
        Element := Element + CellSeparation;
    end;
end;

for CellElements := NumElInCellSHR2 + 1 to NumElInCellSHR1 - 1 do
begin
    Index := CellElements * NumberOfCells;
    CosTerm := CosTable^[Index];
    SumTerm := SinTable^[Index] + CosTerm;
    DifTerm := SinTable^[Index] - CosTerm;
    Element := CellElements;
    while Element < NumPoints do
    begin
        { Combine the X[Element] with the element in }
        { the identical location in the next cell }
        ElementInNextCell := Element + NumElementsInCell;
        Dummy1 := (XReal^[ElementInNextCell] +
                    XImag^[ElementInNextCell]) * CosTerm;
        Dummy2 := XReal^[ElementInNextCell] * DifTerm;
        RealDummy := Dummy1 - XImag^[ElementInNextCell] * SumTerm;
        ImagDummy := Dummy1 + Dummy2;
        XReal^[ElementInNextCell] := XReal^[Element] - RealDummy;
        XImag^[ElementInNextCell] := XImag^[Element] - ImagDummy;
        XReal^[Element] := XReal^[Element] + RealDummy;
        XImag^[Element] := XImag^[Element] + ImagDummy;
        Element := Element + CellSeparation;
    end;
end; { for }

{ Special case: RootOfUnity = EXP(-i PI/2) }
if Term > 1 then
begin
    Element := NumElInCellSHR1;
    while Element < NumPoints do
    begin
        { Combine the X[Element] with the element in }
        { the identical location in the next cell }
        ElementInNextCell := Element + NumElementsInCell;
        RealDummy := XImag^[ElementInNextCell];
        ImagDummy := -XReal^[ElementInNextCell];
        XReal^[ElementInNextCell] := XReal^[Element] - RealDummy;
        XImag^[ElementInNextCell] := XImag^[Element] - ImagDummy;
        XReal^[Element] := XReal^[Element] + RealDummy;
        XImag^[Element] := XImag^[Element] + ImagDummy;
        Element := Element + CellSeparation;
    end;
end;

for CellElements := NumElInCellSHR1 + 1 to
    NumElementsInCell - NumElInCellSHR2 - 1 do
begin
    Index := CellElements * NumberOfCells;
    CosTerm := CosTable^[Index];
    SumTerm := SinTable^[Index] + CosTerm;
    DifTerm := SinTable^[Index] - CosTerm;
    Element := CellElements;
    while Element < NumPoints do
    begin
        { Combine the X[Element] with the element in }
        { the identical location in the next cell }
        ElementInNextCell := Element + NumElementsInCell;
        Dummy1 := (XReal^[ElementInNextCell] +
                    XImag^[ElementInNextCell]) * CosTerm;
        Dummy2 := XReal^[ElementInNextCell] * DifTerm;
        RealDummy := Dummy1 - XImag^[ElementInNextCell] * SumTerm;

```

```

    ImagDummy := Dummy1 + Dummy2;;
    XReal^[ElementInNextCell] := XReal^[Element] - RealDummy;
    XImag^[ElementInNextCell] := XImag^[Element] - ImagDummy;
    XReal^[Element] := XReal^[Element] + RealDummy;
    XImag^[Element] := XImag^[Element] + ImagDummy;
    Element := Element + CellSeparation;
end;
end; { for }

{ Special case: RootOfUnity = EXP(-i 3PI/4) }
if Term > 2 then
begin
    Element := NumElementsInCell - NumElInCellSHR2;
    while Element < NumPoints do
    begin
        { Combine the X[Element] with the element in }
        { the identical location in the next cell }
        ElementInNextCell := Element + NumElementsInCell;
        RealDummy := -RootTwoOverTwo * (XReal^[ElementInNextCell] -
                                         XImag^[ElementInNextCell]);
        ImagDummy := -RootTwoOverTwo * (XReal^[ElementInNextCell] +
                                         XImag^[ElementInNextCell]);
        XReal^[ElementInNextCell] := XReal^[Element] - RealDummy;
        XImag^[ElementInNextCell] := XImag^[Element] - ImagDummy;
        XReal^[Element] := XReal^[Element] + RealDummy;
        XImag^[Element] := XImag^[Element] + ImagDummy;
        Element := Element + CellSeparation;
    end;
end;

for CellElements := NumElementsInCell - NumElInCellSHR2 + 1 to
    NumElInCellLess1 do
begin
    Index := CellElements * NumberOfCells;
    Costerm := CostTable^[Index];
    SumTerm := SinTable^[Index] + Costerm;
    DifTerm := SinTable^[Index] - Costerm;
    Element := CellElements;
    while Element < NumPoints do
    begin
        { Combine the X[Element] with the element in }
        { the identical location in the next cell }
        ElementInNextCell := Element + NumElementsInCell;
        Dummy1 := (XReal^[ElementInNextCell] +
                   XImag^[ElementInNextCell]) * Costerm;
        Dummy2 := XReal^[ElementInNextCell] * DifTerm;
        RealDummy := Dummy1 - XImag^[ElementInNextCell] * SumTerm;
        ImagDummy := Dummy1 + Dummy2;;
        XReal^[ElementInNextCell] := XReal^[Element] - RealDummy;
        XImag^[ElementInNextCell] := XImag^[Element] - ImagDummy;
        XReal^[Element] := XReal^[Element] + RealDummy;
        XImag^[Element] := XImag^[Element] + ImagDummy;
        Element := Element + CellSeparation;
    end;
end; { for }
end;

{-----}
{- Divide all the values of the transformation -}
{- by the square root of NumPoints. If taking the -}
{- inverse, conjugate the output. -}
{-----}

{If Inverse then } {Mod: commented line this out}
    ImagDummy := -1 / Sqrt(NumPoints);
{else
    ImagDummy := 1 / Sqrt(NumPoints)} {Mod: commented these lines out}

    RealDummy := ABS(ImagDummy);

If In erse then {Mod: added this line}
for Element := 0 to NumPoints - 1 do
begin
    XReal^[Element] := XReal^[Element] * RealDummy;
    XImag^[Element] := XImag^[Element] * ImagDummy;
end;
end; { procedure FFT }

```

B1.4 SONAGRAM.PAS

```
UNIT SONAGRAM;
INTERFACE
USES BCRTYPES;

Procedure GetSonagram (CallName,SpecName,SonaName: NameStr);
{
  Executes the SONAGRAM process.
  'CallName' is the base name of the call, 'SpecName' is the input .SF file
  suffix, 'SonaName' is the output Sonagram Storage (.SS) file suffix.

  Produces a file of sonagram drawing information, which can be rendered on the
  screen by the DRAWER process or other used by other graphics programs.

  The programmer may wish to alter certain constants: 'ScreenLength' should
  relect the number of horizontal pixels to be used on the display where the
  sonagram is to be rendered. 'MaxLineSegments' is the maximum number of
  segments permitted per sonagram line (a segment is a portion of a horizontal
  line, one segment is needed for every colour change along the line). If a
  call is found which needs more than 400 segments, the sonagram will be
  incomplete (higher frequencies will overflow off the page) - then
  'MaxLineSegments' should be increased.
}

IMPLEMENTATION
USES Crt,BCDMANIP,GENLIB;

Procedure GetSonagram;

const
  ScreenLength = 600; {No. of points used to display a single sonagram line}
  FreqBase = ScreenLength; {Number of points in 'FreqScale' kHz}
  TimeBase = 200; {Number of points in 'TimeScale' millisecs}
  MaxLineSegments = 400; {The estimated maximum number of segments needed
  to draw a sonagram line}

type
  pointwiselinetype = array [1..ScreenLength] of real;
  linewiselinetype = array [1..MaxLineSegments] of segmenttype;
var
  FreqScale, TimeScale, MaxFactor, MinFactor: real; {Input parameters from BCD}
  SpectralFrame: array [1..MAXFFT div 2] of real;
  Spectral frames: file {of partial SpectralFrame type};
  SamplingFreq: real;
  FftLength, HalfFFT, WindowLength: word;
  NumberOfFrames, frame: longint;
  MaxMag: real;

  LinesInTimeScale, PointsInFreqScale,
  LinesPerMSec, PixelsPerKhZ,
  TimeStretchRatio, FreqStretchRatio: real;
  TransformFrom: 1..MAXFFT;
  TransformTo: 1..ScreenLength;
  BetweenPt: real;
  Finished: boolean;

  LeftPointWiseLine, RightPointWiseLine, TempLeft: ^pointwiselinetype;
  BetweenPointWiseLine, DifferencePoints: pointwiselinetype;
  DifferenceRatio: real;
  LineWiseLine: linewiselinetype;
  Sonagram storage: file {of partial linewiselinetype};
  segment, highestsegnum: 1..MaxLineSegments;
  Level, PreviousLevel: 0..16;
  reallevel: real;
  i: 1..ScreenLength;
  ch: char;

begin
  PutCurrentProcess ('SONAGRAM');

  BCD('read '+CallName+'.ADS Sampling_frequency: real',SamplingFreq);
  BCD('read '+CallName+SpecName+'.SF FFT_length: word',FftLength);
  BCD('read '+CallName+SpecName+'.SF Max_magnitude: real', MaxMag);
  BCD('read '+CallName+SpecName+'.SF '+CallName+SonaName+'.SS Frequency_scale: real',-
FreqScale);
  BCD('read '+CallName+SpecName+'.SF '+CallName+SonaName+'.SS Time scale: real',TimeScale);
  BCD('read '+CallName+SpecName+'.SF '+CallName+SonaName+'.SS Min_factor: real',MinFactor);
  BCD('read '+CallName+SpecName+'.SF '+CallName+SonaName+'.SS Max_factor: real',MaxFactor);

  If MaxFactor < MinFactor then
    GenRunError (0,'GetSonagram error: Max_factor < Min_factor');
```

```

PointsInFreqScale := FreqScale/SamplingFreq * FFtLength;
FreqStretchRatio := FreqBase/PointsInFreqScale;
PixelsPerKHz := FreqBase/FreqScale;
BCD ('write '+CallName+SpecName+'.SF '+
      CallName+SonaName+'.SS PixelsPerKHz: real',PixelsPerKHz);

HalfFFT := FFtLength div 2;

if (HalfFFT)*FreqStretchRatio > ScreenLength
then begin
  TransformFrom := round(ScreenLength/FreqStretchRatio);
  TransformTo := ScreenLength;
end
else begin
  TransformFrom := HalfFFT;
  TransformTo := round(HalfFFT * FreqStretchRatio);
end;

BCD('read '+CallName+SpecName+'.SF Window_length: word', WindowLength);
LinesInTimeScale := TimeScale*SamplingFreq / WindowLength;
TimeStretchRatio := TimeBase/LinesInTimeScale;
LinesPerMsec := TimeBase/TimeScale;
BCD ('write '+CallName+SpecName+'.SF '+CallName+SonaName+'.SS LinesPerMsec: real',LinesPerMsec);

BCD('read '+CallName+SpecName+'.SF Number_of_frames: longint',NumberOfFrames);
writeln ('Calcp plotting ',NumberOfFrames,' Frames.');
```

assign (Spectral_frames,CallName+SpecName+'.SF');

if FileQuery (Spectral_frames,DontAsk) = Exists
 then reset (Spectral_frames, (HalfFFT)*sizeof(SpectralFrame[1]))
 else GenRunError(0,'SF.DAT not found');

assign (Sonagram_storage, CallName+SonaName+'.SS');

if FileQuery (Sonagram_storage, AskOverWrite) in [DoesNotExist, OverWriteOK]
 then rewrite (Sonagram_storage,sizeof(LineWiseLine[1]))
 else halt(0);

new(LeftPointWiseLine);
new(RightPointWiseLine);

highestsegnum := 1;
frame := 0;
BetweenPt := 1;
blockread (Spectral_frames, SpectralFrame, 1);
GraphTransform (SpectralFrame, TransformFrom,
 RightPointWiseLine^,TransformTo);
Finished := FALSE;
Repeat
 If BetweenPt >= 1 then
 begin
 if eof (Spectral_frames) then Finished := TRUE
 Else begin
 TempLeft := LeftPointWiseLine;
 LeftPointWiseLine := RightPointWiseLine;
 RightPointWiseLine := TempLeft;

blockread (Spectral_frames, SpectralFrame, 1);
GraphTransform (SpectralFrame, TransformFrom,
 RightPointWiseLine^,TransformTo);
for i := 1 to ScreenLength do
 DifferencePoints[i] := RightPointWiseLine^[i]-LeftPointWiseLine^[i];

BetweenPt := BetweenPt-1;
inc(frame);
writexy (70,1, Strung(0,0, Frame)+' ');
End{Else};
End{If};

If not Finished then
begin
 DifferenceRatio := Frac(BetweenPt);
for i := 1 to ScreenLength do BetweenPointWiseLine[i] :=
 LeftPointWiseLine^[i] + DifferenceRatio * DifferencePoints[i];

segment := 1;
PreviousLevel := 0;
For i := 1 to ScreenLength do {find and store the segments}
begin
 reallevel := (BetweenPointWiseLine[i]-MaxMag*MinFactor)/
 (MaxMag*(MaxFactor-MinFactor));
if reallevel < 0 then reallevel := 0;
if reallevel >= 1
 then Level := 15
 else Level := trunc (16*reallevel);

{The above quantizes BetweenPointWiseLine[i] to 16 levels. The range BetweenPointWiseLine[i] = MaxMag*MinFactor..MaxMag*MaxFactor is quantized linearly to the range Level = 0..15. When BetweenPointWiseLine[i] is greater than MaxMag*MaxFactor, or less than MaxMag*MinFactor, Level remains at 0 or 15, ie these are saturation, or clipping, points. Thus MinFactor = 0.0, MaxFactor = 0.5 means that the lower half of the actual range of the signal will be linearly quantized to 16 levels, and anything in the higher half of the range will be clipped to level 15.)

```

If (Level <> PreviousLevel) or (i = ScreenLength) then
begin
  LineWiseLine[segment].y := i;
  LineWiseLine[segment].level := PreviousLevel;
  PreviousLevel := Level;
  if segment < MaxLineSegments then inc(segment);

  {if segment >= MaxLinesegments then the sonagram line will be}
  {incomplete but no error message is generated. Make }
  {MaxLineSegments bigger to correct this problem if it occurs}

End{If};
End{For};

With LineWiseLine[segment-1] do level := level+16; {end of line marker}

{segment-1 is used because the loop above usually exits with}
{segment pointing to the last + 1, except when segment}
{equals MaxLineSegments, in which case the last segment}
{is meaningless anyway, so always use the last but one.}

blockwrite (Sonagram_storage, LineWiseLine, segment-1);

If segment > highestsegnum then highestsegnum := segment;

  BetweenPt := BetweenPt + 1/TimeStretchRatio;
End{If}
Until (Finished);

dispose (RightPointWiseLine);
dispose (LeftPointWiseLine);

close (Spectral_frames);
close (Sonagram_storage);
End{Procedure};

END{UNIT}.

```

B1.5 PEAKPICK.PAS

```
UNIT PEAKPICK;
INTERFACE
USES BCRTYPES;

Procedure PickPeaks(CallName, SpecName, PeakName: NameStr);
{
  Executes the PEAKPICK process.
  Extracts peaks from a Spectral Frames (.SF) file and stores in a
  Linked Peak Frames (.LPF) file. The maximum number of peaks is given
  by 'PeaksPerFrame' which is a constant set in the BCRTYPES unit.

  'CallName' is the base bird call name, 'SpecName' is the suffix of the
  .SF file and 'PeakName' .LPF

  For clarity, note: Besides The .SF file, the program requires a
  Single Component (.SC) file of the same name as the .SF file. This file is
  read in near the start of processing, and not accessed from disk afterwards.
  The Spectral_frames file handle is therefore first used to access the .SC
  file and then reused for the .SF file.
}
IMPLEMENTATION
USES Crt, BCDMANIP, GENLIB;

Procedure PickPeaks;

var
  SampFreq, MaxMag, BackMag: real;
  FftLength: word;
  frame, NumberOfFrames: longint;
  Spectral_frames: file;
  SpectralFrame: array[1..MAXFFT div 2] of real;
  ComponentSpectrum: array[0..MAXFFT] of real; {See definition of}
                                              {SpectralFrame in SPECTRA}

  SignalToNoise: real;
  peak: peakframe;
  Peak_frames: file of peakframe;
  i: 1..MAXFFT div 2;
  j: 1..PeaksPerFrame;
  maxnum: 0..PeaksPerFrame;
  peakpos: 0..MAXFFT div 2;
  SpectrumSum, component, SigThresh, max: real; {same type as SpectralFrame[1]}

{}Procedure SubtractPeak (peakposition: word);
{}
{} [Subtract the assumed spectrum of a single component, as stored in
{} ComponentSpectrum, from the spectral frame, at the position specified
{} by 'peakposition', by shifting the ComponentSpectrum peak and
{} scaling it to that of the component at 'peakpos' and then subtracting]
{}
{} Note that the magnitude of the peak in ComponentSpectrum is 1, it resides
{} at the 'FftLength/2' position in the array, and the shifting is achieved
{} by manipulation of the index rather than physically shifting.
{}
{}var
{} disp, i: word;
{} scale: real;
{}begin
{}  disp := FftLength div 2 - peakposition;
{}  scale := SpectralFrame[peakposition];
{}
{}  for i := 1 to FftLength div 2 do
{}    SpectralFrame[i] := SpectralFrame[i] - scale * ComponentSpectrum[i + disp];
{}End{}Procedure;

{}Procedure SortFreqs(var pf: peakframe);
{}
{} Sorts the peaks in the frame so that the lowest frequency appears in
{} in the first element, highest in the last.
{}
{}var
{} i : 1..PeaksPerFrame;
{} temp: peaktype;
{} noswaps: boolean;
{}begin
{}  repeat
{}    noswaps := TRUE;
{}    for i := 1 to PeaksPerFrame-1 do
{}      If pf[i].freq > pf[i+1].freq then
{}        begin
{}          temp := pf[i];
{}          pf[i] := pf[i+1];
{}          pf[i+1] := temp;

```

```

{}      noswaps := FALSE;
{}      End{If};
{}      until noswaps
{}End{SortFreqs};

begin
  PutCurrentProcess ('PEAKPICK');

  BCD('read '+CallName+'.ADS Sampling frequency: real', Sampfreq);
  BCD('read '+CallName+SpecName+'.SF FFT_length: word', FFTlength);
  BCD('read '+CallName+SpecName+'.SF Number of frames: longint', NumberOfFrames);
  BCD('read '+CallName+SpecName+'.SF Background_mag: real', BackMag);
  BCD('read '+CallName+SpecName+'.SF '+CallName+PeakName+'.LPF'+ ' snr: real', SignalToNoise);

  assign (Spectral_frames, CallName+SpecName+'.SC');
  {$I-} reset (Spectral_frames, sizeof(SpectralFrame[1])*FFTlength); {$I+}
  if IOresult <> 0
  then runerror(UserMessage('Error opening Single Component (.SC) file'));

  blockread (Spectral_frames, ComponentSpectrum, 1);
  close (Spectral_frames);

  assign (Spectral_frames, CallName+SpecName+'.SF');
  reset (Spectral_frames, sizeof(SpectralFrame[1])*FFTlength div 2);
  assign (Peak_frames, CallName+PeakName+'.LPF');
  If FileQuery(Peak_frames, AskOverWrite) in [OverWriteOK, DoesNotExist]
  then rewrite (Peak_frames) else halt(255);

  writeln ('Picking peaks from ',NumberOfFrames,' frames');
  frame := 1;
  SigThresh := BackMag*SignalToNoise;

  While not eof(Spectral_frames) do {each frame}
  begin
    writexy (70,1,Strung(0,0,frame));
    blockread (Spectral_frames, SpectralFrame, 1);

    fillchar (peak, sizeof(peak), #0);

    For j := 1 to PeaksPerFrame do {find each peak}
    begin
      peakpos := 0;

      For i := 1 to FftLength div 2 do {each component}
      begin
        component := SpectralFrame[i];
        If (component > SigThresh) and (component > peak[j].mag) then
        begin
          peak[j].mag := component;
          peak[j].freq := i/Fftlength*SampFreq;
          peakpos := i;
        End{If};
      End{For};

      if peakpos <> 0 then SubtractPeak (peakpos);
    End{For};

    SortFreqs(peak);
    write(Peak_frames, peak);
    inc(frame);
  End{While};

  close (Peak_frames);
  close (Spectral_frames);
End{Procedure};
END{UNIT}.

```

B1.6 BACKGND.PAS

```
UNIT BACKGND;
INTERFACE
USES BCRTPES;

Procedure BackGround (CallName, SpecName: NameStr;
                     startframe, endframe: longint);
{
  Executes the BACKGND process.
  Finds the average of all the components in the input Spectral Frames (.SF) file
  between frame number 'startframe' and frame number 'endframe' inclusive.
  Writes the result to the "Backgruond_mag:" field in the Bird Call Descriptor

  This process is designed to be called from another BCR system process,
  due to the interaction required to determine startframe and endframe. Thus the
  currently executing process is saved at the start and restored at the
  end. BACKGND is currently called from DRAWER.

  'CallName' is the base bird name, 'SpecName' is the .SF file suffix.
}

IMPLEMENTATION
USES BCDMANIP;

Procedure BackGround;
var
  Spectral_frames: file;
  SpectralFrame: array [1..MAXFFT div 2] of real;
  Fftlength: word;
  frame: longint;
  framesum, averagesum: real;
  i: 1..MAXFFT div 2;
  curprocsave: string;
begin
  GetCurrentProcess (curprocsave);
  PutCurrentProcess ('BACKGND');
  BCD ('read '+CallName+SpecName+'.SF FFT_length: word',Fftlength);

  assign (Spectral_frames,CallName+SpecName+'.SF');
  reset (Spectral_frames, (FftLength div 2)*sizeof(SpectralFrame[1]));
  seek(Spectral_frames, startframe-1);

  averagesum := 0;

  If startframe < endframe then
  begin
    frame := startframe-1;
    While (not eof (Spectral_frames)) and (frame < endframe) do
    begin
      blockread (Spectral_frames,SpectralFrame,1);
      inc(frame);

      framesum := 0;
      for i := 1 to FftLength div 2 do framesum := framesum + SpectralFrame[i];

      averagesum := averagesum + framesum/(Fftlength div 2);
    End{While};
  End{If};

  averagesum := averagesum/(frame-startframe+1);
  BCD('write '+CallName+SpecName+'.SF Background_mag: real',averagesum);

  PutCurrentProcess (curprocsave);
End{Function};
END.{UNIT}
```

B1.7 CLASSIFY.PAS

```
UNIT CLASSIFY;
INTERFACE
USES BCRTYPES;

procedure GetFrameClasses(CallName, SpecName, PeakName, ClassName: NameStr);
{
  Executes the CLASSIFY process.
  'CallName' is base bird file name, 'SpecName' is Spectral Frames (.SF)
  suffix, 'PeakName' is the input Linked Peak Frames (.LFF) suffix,
  'ClassName' is the output Classified Frames (.CF) suffix. The Spectral
  Frames suffix is used to find required Bird Call Descriptor parameters.
}

IMPLEMENTATION
USES Crt, BCDMANIP, GENLIB;

Procedure GetFrameClasses;

var
  HighPassFreq, DistanceThresh, ClassMagThresh, BackGroundMag: real;
  Peak_frames: file of peakframe;
  Classified_frames: file of classifiedframe;
  frame: peakframe;
  classification: classifiedframe;
  freqs, mags: array[1..PeaksPerFrame] of real;
  freqmean, freqstdv, diffmean, diffstdv: real;
  i: 1..PeaksPerFrame;
  j: 1..PeaksPerFrame+1;
  PeakCount: 0..PeaksPerFrame;
  frameno: longint;

begin
  PutCurrentProcess('CLASSIFY');
  assign (Peak_frames, CallName+PeakName+'.LFF');
  assign (Classified_frames, CallName+ClassName+'.CF');

  if FileQuery (Classified_frames, AskOverWrite) in [OverWriteOK, DoesNotExist]
    then rewrite(Classified_frames)
    else halt(255);

  if FileQuery (Peak_frames, DontAsk) = Exists
    then reset (Peak_frames)
    else GenRunError(0, 'CLASSIFY couldn't find the Linked peak frames file');

  BCD('read '+CallName+PeakName+'.LFF '+CallName+ClassName+'.CF High_pass_freq: real', -
HighPassFreq);
  BCD('read '+CallName+PeakName+'.LFF '+CallName+ClassName+'.CF Distance_thresh:
real', DistanceThresh);
  BCD('read '+CallName+PeakName+'.LFF '+CallName+ClassName+'.CF Class_mag_thresh:
real', ClassMagThresh);
  BCD('read '+CallName+SpecName+'.SF Number_of frames: longint', frameno);
  BCD('read '+CallName+SpecName+'.SF Background_mag: real', BackGroundMag);
  writeln ('Classifying ', frameno, ' frames ...');
  frameno := 1;

  While not eof(Peak_frames) do
  begin
    WriteXY(70, 1, Strung(0, 0, frameno) + ' ');
    read(Peak_frames, frame);

    {Remove Peaks that are below High_pass_freq, and count the peaks}
    {note the peaks have been ordered in ascending frequency}

    PeakCount := 0;
    j := 1;
    {Note this must be boolean short circuit to prevent range check error -}
    {however operation will be OK with range checking off}
    While (j <= PeaksPerFrame) and (frame[j].freq < HighPassFreq) do inc (j);
    if j <= PeaksPerFrame
    then begin
      PeakCount := PeaksPerFrame - j + 1;
      for i := j to PeaksPerFrame do
        frame[i-j+1] := frame[i]
      end;
    end;
  end;
```

```

If PeakCount > 0 then
begin
  for i := 1 to PeakCount do    {Takes up more time than if the stats}
  begin                        {calculations were done directly on}
    freqs[i] := frame[i].freq; {the frame array instead of via the}
    mags[i] := frame[i].mag    {general routine}
  end{For};
  freqmean := Mean(freqs, PeakCount, mags, TRUE);
  freqstdv := StdDev(freqs, freqmean, PeakCount, mags, TRUE);
  classification.mag := Mean(mags, PeakCount, mags, FALSE);
end {If};

If (PeakCount <= 0) or
(classification.mag < ClassMagThresh*BackGroundMag) then
begin
  freqmean := 0;
  classification.mag := 0;
end {If};

if (PeakCount > 1) and (classification.mag <> 0)
{second condition is just to save time}
then begin
  {get the distance between adjacent peaks. The first distance is that
  between the first peak and 0. The first weight is the magnitude of
  the first peak - since there is no magnitude defined for 0 Hz}
  for i := 2 to PeakCount do
  begin
    freqs[i] := frame[i].freq-frame[i-1].freq;
    mags[i] := sqrt(frame[i].mag*frame[i-1].mag);
  end;
  diffmean := Mean(freqs, PeakCount, mags, TRUE);
  diffstdv := StdDev(freqs, diffmean, PeakCount, mags, TRUE);
end;

{you could initialise the means and stdvs so that if the above calculations}
{are not performed, they will still be correctly defined below, but in      }
{in fact they are not used in those cases anyway.                          }

if freqmean = 0
then classification.class := silent
else if freqstdv < DistanceThresh
then classification.class := single
else if diffstdv < DistanceThresh
then classification.class := harmo
else classification.class := multi;

if classification.class = harmo then
begin
  {check that the first or second peak (allowing the first to be
  noise) is either the fundamental or the first harmonic - otherwise
  this can't really be harmonic element}
  classification.freq := diffmean; {the supposed fundamental frequency}
  if not ((abs(freqs[1]-classification.freq) < DistanceThresh) or
    (abs(freqs[2]-classification.freq) < DistanceThresh) or
    (abs(freqs[1]-2*classification.freq) < DistanceThresh) or
    (abs(freqs[2]-2*classification.freq) < DistanceThresh))
  then classification.class := multi;
end;

if classification.class <> harmo then classification.freq := freqmean;

{multiple unrelated frequency frames are noise if they have many peaks}
if (classification.class = multi) and (PeakCount >= PeaksPerFrame-1)
then classification.class := noise;

write (Classified_frames,classification);
inc(framecnt);
end{While};

close(Peak_frames);
close (Classified_frames);
end{Procedure};

END{UNIT}.

```

B1.8 ELEMENT.PAS

```
UNIT ELEMENT;  
INTERFACE  
USES BCRTYPES;
```

```
procedure DetermineElements(CallName, SpecName, ClassName, ElementName: NameStr);  
{  
  Executes the ELEMENT process.  
  Determines the starting points and durations of the elements based on  
  information in the input Classified Frames (.CF) file. Outputs a string of  
  element symbols containing class, frequency and duration and a  
  breakdown of the class content, stored in the Elements (.E) file.  
  
  'CallName' is the base bird name, 'SpecName' is the suffix of the  
  Spectral Frames (.SF) file from which the the input .CF file (suffix given  
  by 'ClassName') was indirectly produced, 'ElementName' is the suffix of  
  the output .E file.  
}
```

```
IMPLEMENTATION  
USES crt, BCDMANIP, GENLIB;
```

```
Procedure DetermineElements;
```

```
var  
  Classified frames: file of classifiedframe;  
  Elements: file of elementtype;  
  ClassFrame: classifiedframe;  
  WindowLength: word;  
  SamplingFreq: real;  
  frameno, NumberOfFrames: longint;  
  ElemFrameCount: word;  
  ElemFreqSum, ElemMagSum: real;  
  BreakDown: array[single..noise] of real;  
  currentsilent: boolean; {Says whether the last frame class was silent or not}  
  DominanceRatio: real;  
  
{DominanceRatio is the Parameter that determines what kind of majority the  
  dominant class needs to have. The closer to one, the more elements will be  
  classified 'mixed' and thus need further processing, resulting eventually in  
  a more detailed (possibly unnecessarily so) representation. Note that if  
  DominanceRatio is less than 0.5, this could result in two classes occuring  
  equally, but one being chosen as dominant}
```

```
{ }procedure WriteElement (class: classtype; freq:real);  
{ }  
{ } Finds the element duration and writes it together with the passed in  
{ } information and the info in 'BreakDown' to the elements file.  
{ } Clears all the variables used to accumulate information about the current  
{ } element, switches the currentsilent state, since it always called at the  
{ } transition of silent to non-silent and vice versa.  
{ }  
{ }var  
{ } Element: elementtype;  
{ } i: single..noise;  
{ }begin  
{ }  
{ } Element.freq := freq;  
{ } Element.class := class;  
{ } for i := single to noise do Element.occurrence[i] := BreakDown[i];  
{ } Element.durn := ElemFrameCount*Windowlength/SamplingFreq;  
{ }  
{ } Write(Elements,Element);  
{ }  
{ } currentsilent := not currentsilent;  
{ } ElemFrameCount := 0;  
{ } FillChar(BreakDown,sizeof(BreakDown),#0);  
{ } ElemFreqSum := 0;  
{ } ElemMagSum := 0;  
{ }  
{ }End;
```

```
{ }procedure CountNonSilent;  
{ }  
{ } increments the accumulator variables to register the occurrence of a  
{ } non-silent frame, counts the frame into the total for the element, and  
{ } adds the frame's magnitude to the total for the element.  
{ }  
{ }begin  
{ } inc(ElemFrameCount);  
{ } (*{} inc(BreakDown[classframe.class]);*)  
{ } With ClassFrame do  
{ } begin
```

```

{} BreakDown[class] := BreakDown[class] + mag;
{} ElemFreqSum := ElemFreqSum + freq;
{} ElemMagSum := ElemMagSum + mag;
{} End;
{}end;

{}procedure WriteNonSilent;
{}
{} Processes the currently accumulated information on a non silent element -
{} the fraction occupied by each class is obtained by normalising the
{} BreakDown magnitude sums to the overall magnitude sum of the element.
{} The class that occurs most often is checked against dominance ratio to see
{} if it has an appropriate majority, thus deciding the class of the element.
{}
{}
{}var
{} i,maxclass: classtype;
{} maxfract: real;
{}begin
{} maxfract := 0;
{} For i := single to noise do
{} begin
{} BreakDown[i] := BreakDown[i]/ElemMagSum;
{} if BreakDown[i] > maxfract
{} then begin
{} maxfract := BreakDown[i];
{} maxclass := i;
{} end;
{} End;
{} if maxfract >= DominanceRatio
{} then WriteElement(maxclass,ElemFreqSum/ElemFrameCount)
{} else WriteElement(mixed,ElemFreqSum/ElemFrameCount);
{}
{}End {Procedure};

begin
PutCurrentProcess('ELEMENT');
assign (Elements,CallName+ElementName+'.E');
assign (Classified_frames, CallName+ClassName+'.CF');

if FileQuery (Elements, AskOverWrite) in [OverWriteOK, DoesNotExist]
then rewrite(Elements)
else halt(255);

if FileQuery (Classified_frames, DontAsk) = Exists
then reset (Classified_frames)
else runerror(UserMessage('ELEMENT couldn't find the Classified frames file'));

BCD('read '+CallName+SpecName+'.SF Window length: word',Windowlength);
BCD('read '+CallName+'.ADS Sampling frequency: real',SamplingFreq);
BCD('read '+CallName+SpecName+'.SF Number_of_frames: longint',NumberOfFrames);
BCD('read '+CallName+ElementName+'.E Dominance_ratio: real',DominanceRatio);

ElemFrameCount := 0;
FillChar(BreakDown,sizeof(BreakDown),#0);
ElemFreqSum := 0;
ElemMagSum := 0;

writeln ('Finding elements from ',NumberOfFrames,' frames');

if not eof(Classified_frames) then read(Classified_frames,classframe);
currentsilent := (classframe.class = silent);
if currentsilent then inc (ElemFrameCount) else CountNonSilent;

{
This ensures that the algorithm is "expecting" the first (partial) element
correctly - it will not write the element as soon as it sees the first
frame, but will wait until the transition from the class of the first frame
to either silent or non-silent, as appropriate.

The first element is not useful for recognition, but for matching the
element representation up with the real one. It can easily be ignored by
the syntax checker.
}

reset(Elements);
frameno := 1;
While not eof(Classified_frames) do
begin
WriteXY (70,1,Strung(0,0,frameno)+' ');
read(Classified_frames, classframe);

```

```

    if currentsilent
    then if classframe.class = silent
        then inc(ElmFrameCount) {ie count silent}
        else begin
            WriteElement(silent,0); {ie write silent}
            CountNonSilent;
        end
    else if classframe.class = silent
        then begin
            WriteNonSilent;
            inc(ElmFrameCount); {ie count silent}
        end
        else CountNonSilent;

    inc(frameno);
End; {While}

if classframe.class = silent then WriteElement(silent,0)
    else WriteNonSilent;
{
    This handles the last element. If the current frame was the same type as
    before, a partial element is written. If it was of a type that caused a
    transition, the full element will have been written already, and the last
    element will just be the last frame. Thus there will always be an
    incomplete element at the end of the sequence that the syntax checker can
    confidently ignore.
}

close (Elements);
close (Classified_frames);
End{Procedure};

END{UNIT}.

```

B1.9 SYNTAX

The SYNTAX listing has been provided twice, in a simplified and a full form. The simplified form contains only that code which is essential to the algorithm, and should be used when following the discussion in Appendix A. The full listing includes the tracing and debugging and other optional code that makes the listing harder to read.

B1.9.1 Simplified SYNTAX.PAS

```
{
  This is SYNTAX.ESS, the essential form of the SYNTAX software. It is the
  source code that would be compiled if the symbols Trace, Debug, TrackResult
  and MachineResult are all left undefined in SYNTAX.PAS. This leaves the
  only the part of the program that does the syntax checking and outputs the
  result in human readable form. Use this source listing in following the
  major part of the explanation of the SYNTAX module in appendix A of the
  document "Computer-aided bird call recognition".
}

UNIT SYNTAX;

INTERFACE
USES BCRTypes;

procedure CheckSyntax (ElemFileName, CandFileName: string);
{
  Executes the SYNTAX process

  Identifies an Elements file ('ElemFileName') by comparing it to the
  syntax specs listed in the text file named 'CandFileName'. Output (either
  a table of results (in human or machine readable form) or a detailed
  track of each comparison) goes to standard output. This can be redirected
  to a file, and probably should be if Trace is enabled.

  The machine readable format is similar to the human format - but there are
  no labels or delimiters or text of any kind, and instead of scores being
  left blank when essential elements are not found, a dummy score is inserted.
  For the Match Count, 0 is inserted (since a low match count is "bad") and
  for the Failure Count, 1000 is inserted to simulate a high number of failures.
  The Y/N indication of whether essentials were present is left out. Example:

  Human format
  -----
  <...File Headings....>
  <.....Table headings.....>
  =====
  Bird1      Y      3.7      6      65
  Bird2      N      1.2
  Machine format
  -----
  3.7      6      65
  1.2      1000      0

  The user can feed the output to a plotting package, or some other program that
  extracts the data for its own purposes.
}

IMPLEMENTATION
USES
  GENLIB, OBJLIB;

VAR
  stdout: text; {The initialisation section of this unit assigns}
               {stdout to '', which give the standard output device}

TYPE
  IdString = string[30]; {String used to identify element specs and}
                       {syntax tree nodes}
  IdStringPtr = ^IdString;

{=====OBJECT DEFINITIONS==}

{=====SpecFileTextlines OBJECT==}
{
  File object for use with SpecFile (see SyntaxTree, ElemSpecListItem).
  Expected returns a useful error message including the line number at which
  it occurred.
}
{=====}

TYPE
  SpecFileTextLines = object (Textlines)
    function Expected (s: string) :string;
  End;
```

```

Function SpecFileTextLines.Expected;
begin
  Expected := 'Error in '+TheFileName+', line '+ Strung(0,0,LinesRead)+
    ': '+s+' expected';
End;

{=====TracedList OBJECT==}
{
  A List with:

  Init - does the normal List.Init, but also a initialises the list's
  window to as specified by the parameters. If the fore- and background
  colours are the same, this is assumed to mean that they should not be
  changed from their current values.

  Show - displays the list based on a ShowItem virtual method. The current
  item is shown highlighted. The list is shown in a TextWindow specified
  by the passed-in 'Win' object.

  ShowItem - must do some kind of write (not writeln) using the current item.
}
{=====}
TYPE
{$ifndef trace}
  TracedList = List;
{$else}
  TracedList = object (List)
    procedure ShowItem; virtual;
    procedure Show (var Win: TextWindow);
  End; {TracedList}

Procedure TracedList.ShowItem;
begin
End; {TracedList.ShowItem}

Procedure TracedList.Show;
var
  CurrentSave: ListItemPtr;
  CurrentSeen: boolean;
  lines: byte;
  lightfore: boolean;
begin
  CurrentSave := Current;
  Win.Activate;
  clrscr;

  lines := 1;
  CurrentSeen := FALSE;
  SetToHead;
  SetToNext;
  While not (AtTail or (CurrentSeen and (lines > Win.d))) do
  begin
    if lines > Win.d
    then begin
      gotoxy(1,1);
      Deline;
      gotoxy(1,Win.d);
    end
    else gotoxy (1,lines);
    if Current = CurrentSave
    then begin
      lightfore := Win.Fcol in [White];
      textbackground (White*byte(lightfore));
      textcolor (White*byte(not lightfore));
      CurrentSeen := TRUE;
      ShowItem;
      textcolor(Win.FCol);
      textbackground(Win.BCol);
    end
    else ShowItem;
    inc (lines);
    SetToNext;
  End;

  Win.Deactivate;
  Current := CurrentSave;
End; {TracedList.Show}

{$endif}

```

```

{=====ElementListItem OBJECT==}
{
  Item object for ElementList. The Init loads an elementtype into the 'e'
  field
}
{=====}
TYPE

ElementListItem = Object (ListItem)
  e: elementtype;

  constructor Init (elem: elementtype); {create list from element file}
End; {ElementListItem}

ElemListItemPtr = ^ElementListItem;

Constructor ElementListItem.Init;
begin
  e := elem;
End; {ElementListItem}

{=====ElementList OBJECT==}
{
  List of elements as read in from .E file. Item returns a pointer to the
  current ElemListItem object, without detaching it from the list.
  Init reads in elements from .E file.

  CopyInit is meant to be used for lists that dont actually have their own
  list structure associated with them, but will be used as handles on an
  existing list structure, i.e. they will be assigned to another ElementList.
  For some reason the assignment has no effect if the handle has not been
  initialised, even though no use of virtual methods are implied in the
  process. Strange. When tracing is on, CopyInit creates a separate window
  for the copied list. CopyDone removes the separate window if tracing is on
  but does nothing otherwise.
}
{=====}
TYPE

ElementList = Object (TracedList)
  constructor Init (ElemFileName: string);
  constructor CopyInit (Source: ElementList);
  function Item: ElemListItemPtr;
End; {ElementList}

Constructor ElementList.CopyInit;
begin
  Self := Source;
End;

Constructor ElementList.Init;
var
  efile: file of elementtype;
  e: elementtype;
  i: ElemListItemPtr;
begin
  TracedList.Init;

  assign (efile, ElemFileName);
  {$I-} reset (efile); {$I+ }
  if IOresult = 0
  then begin
    While not eof(efile) do
    begin
      read (efile,e);
      Attach (new(ElemListItemPtr, Init(e)));
    end; {While}
    close (efile);
  end
  else runerror(UserMessage ('Error opening element file '+ElemFileName));
End; {ElementList.Init}

function ElementList.Item;
begin
  Item := ElemListItemPtr(List.Item(FALSE));
End; {ElementList.Item}

```

TYPE

```
{
Because we require delayed compiler-binding of pointer type identifiers,
we had to have everything under one type definition, which meant
separating the methods from the type declarations for the following objects:
SynTreeNodeList/Item, ElemSpecList/Item and SyntaxTree. Sorry.
}

SynTreeNodePtr = ^SynTreeNode;

{-----SynTreeNodeListItem OBJECT--}
{
Item object for SynTreeNodeList type. Init accepts a pointer to a
SyntaxTree Node.
}
{-----}
SynTreeNodeListItem = object (ListItem)
node: SynTreeNodePtr;
constructor Init (p: SynTreeNodePtr);
End; {SynTreeNodeListItem}

SynTreeNodeListItemPtr = ^SynTreeNodeListItem;

{-----SynTreeNodeList OBJECT--}
{
A list of pointers to the nodes in a syntax tree. Used for the 'Nodes' list
in the ElemSpec object and the 'Branches' list in the SynTreeNode record.
Item returns a pointer to the current SyntaxTree node, without detaching.
Append allows another SynTreeNodeList to be inserted into the list at the
current position, without changing that position.
}
{-----}
SynTreeNodeList = object (TracedList)
function Item: SynTreeNodeListItemPtr;
procedure Append (STNList: SynTreeNodeList);
End; {SynTreeNodeList}

{-----ElemSpecListItem OBJECT--}
{
Item object for ElemSpecList.
Init reads in an element spec from a file 'f' which must be open and
positioned at the start (after the label) of an element spec as defined in
a .E file. The identifier of the spec is passed in 'name', and whether it
is essential or not, in 'essentialflag'.
Match compares the current item with a passed in element 'element',
returning 0 if the durations do not match, 1 if the durations match,
2 if the frequencies match as well, 3 if the class occurrences match as well.
Silent returns true if the spec is for a silent element.
}
{-----}
MatchType = 0..4;

ElemSpecListItem = object (ListItem)
identifier: IdString;
Essential: boolean;
freqmin, freqmax: real;
durnmin, durnmax: real;
profile: array[single..noise] of byte;
Nodes: SynTreeNodeList;
OccurrenceDistance: real;

constructor Init (var sf:SpecFileTextLines; name: string;
essentialflag: boolean);
function Match (element: elementtype): MatchType;
function SilentSpec: boolean;
destructor Done; virtual;
End; {ElemSpecListItem}

ElemSpecListItemPtr = ^ElemSpecListItem;

{-----ElemSpecList OBJECT--}
{
List of element specifications - to be matched against the input elements -
as read in from the Element Specification part of the .S file. This object
is only used inside a SyntaxSpec. Item returns a pointer to the current
ElemSpecListItem, without detaching.
}
{-----}
ElemSpecList = object (TracedList)
function Item: ElemSpecListItemPtr;
End; {ElemSpecList}
```

```

{-----SyntaxSpec OBJECT-----}
{
  A list ('ElemSpecs') and a tree. The tree structure's nodes are SynTreeNode,
  which each contain a pointer ('ElemSpec') to an item in the ElemSpecs list,
  and a list 'SynTreeNodeList' of pointers to other tree nodes.
  The two main methods, which take an ElemList object 'Elems' as their
  input, are:

  PreliminaryScore - scores the call by averaging the scores of the matches
  between the 'Elems' items and the 'ElemSpec' items over the number of
  'Elems' items. 'EssPres' is true if all specs marked essential match an
  element in 'Elems'.

  CheckSyntax - scores the call by traversing the syntax tree based on the
  input 'Elems' (ie comparing each element in turn with the 'ElemSpec'
  at the appropriate node and counting the discontinuities in the traversal,
  which is returned in FailureScore. 'MatchScore' is the sum of the match
  levels obtained for each element in the call.

  The Init method creates the ElemSpecs list by reading the headers in the
  ELEMENTS: part of the .S file (whose name is passed to Init as
  'CandidateName') and then building ElemSpecListItems, passing them the
  correct file position in their Init. It builds the syntax tree by reading
  the information under the SYNTAX: part of the .S file. It also reads the
  'BirdName' from the .S file.

  The Done method disposes all the SynTreeNodes and the ElemSpecs list.
}
{-----}
SynTreeNode = record
  Identifier: IdString;
  ElemSpec: ElemSpecListItemPtr;
  Branches: SynTreeNodeList;
end;

PathStr = string[79];
SyntaxSpec = object
  BirdName: string[40];
  ElemSpecs: ElemSpecList;

  constructor Init (CandidateName: PathStr);
  procedure CheckElements (Elems: ElementList;
    var AvgElementScore: real;
    var EssPres: boolean);
  procedure CheckSyntax (Elems: ElementList;
    var Failures, MatchTotal: word);
  destructor Done;
End; {SyntaxSpec}

{=====SegmentTrackItem OBJECT=====}
{
  Item object for the TrackSegment list object. Records a node that matched
  an element, and the match score.
}
{=====}
TYPE
  SegmentTrackItem = Object (ListItem)
    NodeId: IdString;
    Score: MatchType;

    constructor Init (N: IdString; S: MatchType);
  End; {SegmentTrackItem}

  SegmentTrackItemPtr = ^SegmentTrackItem;

Constructor SegmentTrackItem.Init;
begin
  ListItem.Init;
  NodeId := N;
  Score := S;
End;

{=====EntryPointList OBJECT=====}
{
  List of nodes (entry points) in the tree that match a particular element.
  Init finds the entry points for the passed in 'Element', by searching
  through 'Specs' for a match. This is done by appending the Nodes list of
  any Specs item that matches the current element.
}

```

```

SetToLongestSegment is where the comparison between the elements and the
syntax tree actually takes place. It sets the current position of the
passed in ElementList, 'ReturnElementst', to the end (actually one after the
end) of the longest continuous (unbroken by failures to match) segment of
elements, based on the initial current position of 'ReturnElements', the
list of entry points and the syntax tree structure. Also returns the score
(sum of match levels ) of the segment in 'SegmentScore'.
}
=====)
TYPE
  EntryPointList = object (SynTreeNodeList)
    constructor Init (EntryElement: elementtype; Specs: ElemSpecList);
    procedure SetToEndOfLongestSegment
      (var ReturnElements: ElementList;
       var SegmentScore: word);
  End; {EntryPointList}

Constructor EntryPointList.Init;
begin
  TracedList.Init;

  Specs.SetToHead;
  Specs.SetToNext;
  While not Specs.AtTail do with Specs.Item^ do
    begin
      if Match (EntryElement) > 0 then Append (Nodes);
      Specs.SetToNext;
    End; {While}
  End; {EntryPointList.Init}

Procedure EntryPointList.SetToEndOfLongestSegment;
var
  Segment, LongestSegment: ElementList;
  SegmentLength, MaxSegmentLength: word;
  ThisSegmentScore, MaxSegmentScore: word;

()Procedure FindEndOfSegment (SyntaxTreeNode: SynTreeNodePtr);
()
() Tries to follow the syntax tree using the element sequence in 'Segment'.
() The latter's current position is left at the element which first failed to
() match a branch node. The count of elements which succesfully matched nodes
() is placed in 'SegmentLength', and the sum of the match levels of the
() segment's elements is placed in 'SegmentScore'. 'SyntaxTreeNode' is the
() node in the syntax tree where parsing starts, and is assumed to match the
() initial current element in 'Segment'.
()
(){}var
(){}
(){} MatchLevel, MaxMatch: MatchType;
(){} Branch, MaxBranch: SynTreeNodePtr;
(){} Element: elementtype;
(){} PastLabels,           {Used to check for infinite loops in labels}
(){} AllBranches: SynTreeNodeList; {Local copy of the node's branch list with}
(){}                               {any label node's branch lists appended }
(){}
(){}Function NodeSeen (STNList: SynTreeNodeList):boolean;
(){} Checks STNList to see if the node of the item that was current when the
(){} function was called appears previously in the list, returning TRUE if so.
(){}var
(){} tempBAE: boolean;
(){} CheckNode: SynTreeNodeListItemPtr;
(){}begin
(){} tempBAE := FALSE;
(){} CheckNode := STNList.Item;
(){} STNList.SetToHead;
(){} STNList.SetToNext;
(){} While not ((STNList.Item=CheckNode) or tempBAE
(){}           or STNList.AtTail {for safety - shouldn't happen}) do
(){}   begin
(){}     tempBAE := (CheckNode^.node = STNList.Item^.node);
(){}     STNList.SetToNext;
(){}   End; {While}
(){} NodeSeen := tempBAE;
(){}End; {NodeIn}
(){}
(){}begin
(){} {Handle the first element in the segment separately since the loop}
(){} {checks the BRANCHES of the current syntax tree node.           }
(){}
(){} SegmentLength := 1; {The first element is known to match}
(){} ThisSegmentScore := SyntaxTreeNode^.ElemSpec^.Match (Segment.Item^.e);
(){}
(){} {To save this match from being redone you could store a matchlevel in }

```

```

() {in the entrypointlist item and pass it to this sub-proc, but it ain't }
() {worth the effort}
()
() MaxMatch := ThisSegmentScore;
() Segment.SetToNext;
()
() {Go through the segment and the tree, checking if elements match a branch}
() { off nodes}
() While (not Segment.AtTail) and (MaxMatch > 0) do
() begin
()   Element := Segment.Item^.e;
()   MaxMatch := 0;
()   AllBranches.Init;
()   AllBranches.Append (SyntaxTreeNode^.Branches);
()   AllBranches.SetToHead;
()   AllBranches.SetToNext;
()
()   {Go through the syntax tree node's branch list looking for the branch}
()   {that best matches the current Segment element}
()   While not AllBranches.AtTail do
()   begin
()     Branch := AllBranches.Item^.node;
()     if Branch^.ElemSpec = NIL
()     then begin
()       {This is a label node so add its branches to AllBranches}
()       {But first make sure we haven't already had this label node -}
()       {in which case we are going round in circles}
()
()       if not NodeSeen(AllBranches)
()       then AllBranches.Append (Branch^.Branches)
()       else runerror (UserMessage('Self referential label node path encountered'));
()     end
()     else begin {A normal branch node}
()       MatchLevel := Branch^.ElemSpec^.Match (Element);
()
()       if MatchLevel > MaxMatch
()       then begin
()         MaxMatch := MatchLevel;
()         MaxBranch := Branch;
()       end;
()     end;
()
()     AllBranches.SetToNext;
()   end; {While}
()
()   {if a matching branch was found, advance the element and the node}
()   If MaxMatch > 0 then
()   begin
()     inc (SegmentLength);
()     inc (ThisSegmentScore,MaxMatch);
()     Segment.SetToNext;
()     SyntaxTreeNode := MaxBranch;
()   end;
()
()   AllBranches.Done (TRUE); {Destroy the list items}
() end; {While}
()End; {FindEndOfSegment}

begin
  MaxSegmentLength := 0;
  SetToHead;
  SetToNext;
  While not AtTail do
  begin
    {The segment starts off identical to element list}
    Segment.CopyInit (ReturnElements);

    {Now return Segment with current position pointing after the end,}
    {together with SegmentLength and SegmentScore}
    FindEndOfSegment (Item^.node);

    {See if this is the longest segment so far, and preserve it if so}
    If SegmentLength > MaxSegmentLength then
    begin
      SegmentScore := ThisSegmentScore;
      MaxSegmentLength := SegmentLength;
      LongestSegment.CopyInit (Segment);
    end; {If}

    SetToNext;
  end; {While}
end;

```

```

    {Set the element list so that the current position points after the end of}
    {the longest segment. Return the score.}
    ReturnElements := LongestSegment;

End; {EntryPoints.SetToEndOfLongestSegment}

{-----SynTreeNodeListItem METHODS--}
Constructor SynTreeNodeListItem.Init;
begin
    node := p;
End; {SynTreeNodeListItem.Init}

{-----SynTreeNodeList METHODS--}
Function SynTreeNodeList.Item;
begin
    Item := SynTreeNodeListItemPtr(List.Item (FALSE));
End; {SynTreeNodeList.Item}

Procedure SynTreeNodeList.Append;
var
    CurrentSave: ListItemPtr;
begin
    CurrentSave := Current;           {We don't want to change the current pos}
    STNList.SetToHead;                {Note STNList is a local copy so this }
    STNList.SetToNext;                {won't affect the state of the original. }
    While not STNList.AtTail do
    begin
        Attach(new(SynTreeNodeListItemPtr, Init (STNList.Item^.node)));
        STNList.SetToNext;
    end; {While}
    Current := CurrentSave;
End; {SynTreeNodeList.Append}

{-----ElemSpecListItem METHODS--}
Constructor ElemSpecListItem.Init;
var
    line: string;
    seq: StringSeq;
    error: integer;

{}procedure varset (var var1, var2: real; varlabel: string);
{}begin
{}    if UpString(seq.Getword) <> varlabel then
{}        runerror (UserMessage(sf.Expected(''+varlabel+'')));
{}    val (seq.Getword, var1, error);
{}    if error <> 0 then
{}        runerror (UserMessage(sf.Expected(varlabel+'number')));
{}    if (UpString(seq.Getword) <> 'TO') then
{}        runerror (UserMessage(sf.Expected('TO')));
{}    val (seq.Getword, var2, error);
{}    if error <> 0 then
{}        runerror (UserMessage(sf.Expected(varlabel+'number')));
{}end;

var
    i: classtype;
begin
    seq.Init(sf.NextLine);
    if Upstring(seq.Getword) = 'SILENCE'
    then begin
        {fillchar (Self,SizeOf(Self), #0); messed up the object - possibly
        the VMT address got trashed - anyway got error 210 Object not
        initialised when tried executing Done.}
        freqmin := 0;
        freqmax := 0;
        seq.Reset;
        varset (durnmin, durnmax, 'SILENCE');
    end
    else begin
        {Set the spec fields}
        seq.Reset;
        varset (freqmin, freqmax, 'FREQUENCY');
        seq.Init(sf.NextLine);
        varset (durnmin, durnmax, 'DURATION');
        seq.Init (sf.NextLine);
        i := single;
        repeat
            if UpString (seq.GetWord) <> classname[i] then
                runerror (UserMessage(sf.Expected(classname[i])));
            val (seq.Getword, profile[i], error);
            if error <> 0 then runerror (UserMessage(sf.Expected('Number')));
            inc(i);
        until (i > noise);
    end;
end;

```

```

        seq.Init(sf.NextLine);
        if UpString(seq.GetWord) <> 'OCCURRENCE_DISTANCE' then
            runerror(UserMessage(sf.Expected('OCCURRENCE_DISTANCE')));
        val (Seq.GetWord, OccurrenceDistance,error);
        if error <> 0 then runerror(UserMessage(sf.Expected('Number')));
    end;
    identifier := name;
    Essential := essentialflag;

    {Create an empty nodes list - this will be expanded by someone else}
    Nodes.Init;
End; {ElemSpecListItem.Init}

Destructor ElemSpecListItem.Done;
begin
    Nodes.Done (TRUE);
End; {ElemSpecListItem.Done}

Function ElemSpecListItem.SilentSpec;
begin
    SilentSpec := (freqmax = 0) and (freqmin = 0);
    {This is just how it currently signals a silence - it would be }
    {computationally more efficient to have an additional silence or }
    {class flag in the ElemSpecListItem, and to set this flag when }
    {reading in the soec from the .S file. }
End;

Function ElemSpecListItem.Match;
var
    i: classtype;
    sumsqr: real;
begin
    Match := 0;
    if element.class = silent
    then begin
        if ((durnmin = 0) and (durnmax = 0)) {durnmin=durnmax=0 signals that}
            {the spec matches any silence }
        then Match := 4
        else if SilentSpec {Full score if the right}
            {length, otherwise minimum score}
        then if (durnmin <= element.durn) and (element.durn <= durnmax)
            then Match := 4 else Match := 1;
        end
    else if not SilentSpec and
        (durnmin <= element.durn) and (element.durn <= durnmax) then
    begin
        Match := 2; {durations match}
        If ((element.freq >= freqmin) and (element.freq <= freqmax)) then
        begin
            Match := 3; {frequencies match too}

            {Check distance between class occurrence vector }
            { and spec profile vector and give 4 if the distance is small }
            sumsqr := 0;
            for i := single to noise do
                sumsqr := sumsqr+sqr(profile[i]-100*element.occurrence[i]);
            if sqrt(sumsqr) <= OccurrenceDistance then Match := 4;
        End; {If}
    End {If};
End {ElemSpecListItem.Match};

{-----ElemSpecList METHODS--}
Function ElemSpecList.Item;
begin
    Item := ElemSpecListItemPtr(List.Item(FALSE));
End; {ElemSpecList.Item}

{-----SyntaxSpec METHODS--}
Constructor SyntaxSpec.Init;
var
    SpecFile: SpecFileTextLines;
    firstword: string[40];
    seq: StringSeq;
    name: IdString;
    essential: boolean;

{}Procedure SkipToHeading;
{}
{} puts the first word of the next non-blank line in 'firstword', and the
{} entire line is in 'seq' with the current position pointing at the second
{} word. 'firstword' is converted to uppercase.
{}
{}begin
{} repeat

```

```

{} seq.Init(SpecFile.NextLine);
{} firstword := UpString(Seq.GetWord);
{} until firstword <> '';
{}End; {SkipToHeading}

{}Function FindHeading (heading:string): string;
{}
{} Returns the remainder of the line in 'SpecFile' that starts with 'heading'
{} or halts with run time error if that line is not found.
{}
{}begin
{} {SpecFile.Open ('%%Reopen%%'); here to allow headings in any order}
{} {in the .S file}
{} repeat
{}   SkipToHeading
{} until (firstword = UpString(heading)) or (firstword = '%%END');
{} if firstword = '%%END' then
{}   runerror (UserMessage(SpecFile.Expected(heading)));
{} FindHeading := seq.Rest;
{}End; {FindHeading}

{}Procedure CreateElemSpecs;
{}
{} Read element specs from the.S file and build the ElemSpecs list.
{} Stops when "SYNTAX:" is reached.
{}
{}begin
{} ElemSpecs.Init;
{}
{} SkipToHeading;
{} {we have reached an element spec label, so check if its essential }
{} {and then make an item, attach it, skip blanks to the next label until }
{} {the end of ELEMENTS: , ie SYNTAX: is reached. }
{}
{} While not ((firstword = '%%END') or (firstword = 'SYNTAX:')) do
{} begin
{}   name := firstword;
{}   if name[length(name)] = ':'
{}   then essential := (UpString(seq.GetWord) = 'ESSENTIAL')
{}   else runerror (UserMessage(SpecFile.Expected('Colon ":"')));
{}
{}   {Create the spec object and add it to the end of the list}
{}   ElemSpecs.Attach(new(ElemSpecListItemPtr, Init (SpecFile,name,essential)));
{}
{}   SkipToHeading;
{} End; {While}
{} if firstword = '%%END' then
{}   runerror (UserMessage(SpecFile.Expected('SYNTAX:')));
{}End; {CreateElemSpecs}

{}Procedure CreateSyntaxTree;
{}
{} Read node specs from the .S file and buld the syntax tree.
{} Stops when the end of the file is reached.
{}
{}var
{} TempNodeList: SynTreeNodeList; {holds the nodes until they can be linked}
{} {correctly on a second pass}
{} AnotherNodeList: SynTreeNodeList; {Another copy of the TempNodeList}
{} TempId: IdStringPtr;
{}
{}begin
{} TempNodeList.Init;
{}
{} { Remember node spec format is }
{} { <node identifier>: }
{} { <element spec name> <list of branch node identifiers> }
{}
{} {FIRST PASS: READ}
{}
{} {Go through file adding nodes to the temporary list. Force the Branches to}
{} {point to strings containing node names rather than the nodes themselves, }
{} {which may not have been read in yet.}
{}
{} SkipToHeading; {Look for first node}
{} While not (firstword = '%%END') do
{} begin
{}   {Get node identifier}
{}   name := firstword;
{}   if name[length(name)] <> ':'
{}   then runerror (UserMessage(SpecFile.Expected('Colon ":"')));
{}
{}   {Make a new node, attach it and give it its name}
{}   TempNodeList.Attach (new(SynTreeNodeListItemPtr, Init (new(SynTreeNodePtr))));
{}   TempNodeList.Item^.node^.Identifier := name;

```

```

{}
{} {Get node details...}
{} seq.Init (SpecFile.NextLine);
{} if seq.EndOfSequence
{} then runerror (UserMessage(SpecFile.Expected('Element spec identifier')));
{}
{} {... node element spec name...}
{} name := UpString(seq.GetWord)+'.';
{}
{} if name <> '.'
{} then begin {A normal node}
{}   {Search for the element with that identifier in ElemSpecs}
{}   ElemSpecs.SetToHead;
{}   ElemSpecs.SetToNext;
{}   while (not ElemSpecs.AtTail) and (ElemSpecs.Item^.Identifier <> name)
{}   do ElemSpecs.SetToNext;
{}   if ElemSpecs.AtTail then
{}   runerror (UserMessage ('Unknown element spec identifier in line '+
{}     Strung(0,0,SpecFile.LinesRead)+' of '+SpecFile.FileName));
{}
{}   {Set node's element spec pointer to the found element spec...}
{}   TempNodeList.Item^.node^.ElemSpec := ElemSpecs.Item;
{}
{}   {... and install the complementary pointer, ie:
{}   {Add a pointer to the node onto the found element spec's 'Nodes' list}
{}   ElemSpecs.Item^.Nodes.Attach(new(SynTreeNodeListItemPtr,
{}   Init(TempNodeList.Item^.node)));
{}   end
{} else TempNodeList.Item^.node^.ElemSpec := NIL; {A label node}
{}
{} {... initialise the node branch list. Get branch node names and make }
{} {the branches point to the names. The names will be destroyed on the }
{} {second pass, ie when the branches are set to their correct places. }
{} TempNodeList.Item^.node^.Branches.Init;
{} While not seq.EndOfSequence do
{} begin
{}   new(TempId);
{}   TempId^ := UpString(seq.GetWord);
{}   TempNodeList.Item^.node^.Branches.Attach
{}   (new (SynTreeNodeListItemPtr, Init(SynTreeNodePtr(TempId))));
{} End; {While}
{}
{} {Go to next node spec}
{} SkipToHeading;
{} End; {While}
{}
{} {SECOND PASS: RESOLVE}
{}
{} AnotherNodeList.Init;
{} AnotherNodeList := TempNodeList; {Another is a doppelganger - even it's own
{}   {Head and Tail aren't used - so it mustn't}
{}   {Be Done'd - else Temp can't be.}
{}
{} {Go through the temp node list resolving syntax tree node references}
{} {in the nodes' branch lists. }
{} TempNodeList.SetToHead;
{} TempNodeList.SetToNext;
{} While not TempNodeList.AtTail do With TempNodeList.Item^.node^ do
{} begin
{}   {Go through the branches of the current node, matching identifiers with}
{}   {the names in the NodeList (use AnotherNodeList because otherwise you }
{}   {mess up the current position of TempNodeList. }
{}   Branches.SetToHead;
{}   Branches.SetToNext;
{}   While not Branches.AtTail do
{}   begin
{}     TempId := IdStringPtr(Branches.Item^.node);
{}
{}     {Go through the node list looking for nodes with the name pointed to}
{}     {by the current branch. }
{}     AnotherNodeList.SetToHead;
{}     AnotherNodeList.SetToNext;
{}     while (not AnotherNodeList.AtTail) and
{}     (TempId^+'.' <> AnotherNodeList.Item^.node^.Identifier) do
{}     AnotherNodeList.SetToNext;
{}     if AnotherNodeList.AtTail then
{}     runerror (UserMessage ('Unknown node identifier: '+TempId^+
{}       ' in branch list of '+ElemSpec^.Identifier));
{}
{}     {Found the identifier, so make the branch point to that node and }
{}     {destroy the name it had been pointing to. }
{}     {Note that in order to do this efficiently, it was necessary to }
{}     {write to the item objects 'node' field - otherwise we would have }
{}     {had to create a new item object, attach it into the list and }
{}     {detach the old one - a hassle. }
{}     Branches.Item^.node := AnotherNodeList.Item^.node;

```

```

    {}      dispose(TempId);
    {}
    {}      Branches.SetToNext;
    {}      End; {While}
    {}
    {}      TempNodeList.SetToNext;
    {}      End; {While}
    {}
    {} {Destroy the list - the SynTreeNode are not destroyed because that}
    {} {would have to be coded into SynTreeNodeListItem.Done, which I have}
    {} {thoughtfully refrained from doing. The tree of SynTreeNodes is a }
    {} {separate to any list of SynTreeNodes that might be used to reference}
    {} {it, and so should be destroyed by SynTaxTree.Done directly and not}
    {} {by any SynTreeNodeList.Done }
    {}
    {} TempNodeList.Done (TRUE);
    {}End; {CreateSyntaxTree}

begin {Init}

    {Open the .S file}
    {$I-} SpecFile.Open (CandidateName+'.S'); {$I+}
    if IOresult <> 0 then
        runerror (UserMessage('Error opening candidate file: '+CandidateName+'.S'));

    {Find the bird name}
    BirdName := FindHeading('BIRD:');

    {Find the "ELEMENTS:" heading}
    if FindHeading ('ELEMENTS:') = '' then; {Ignore what's after ELEMENTS, just }
        {place the file position}

    {Create the ElemSpecs list from the information under ELEMENTS;}
    CreateElemSpecs;

    {Create syntax tree from information under SYNTAX;}
    CreateSyntaxTree;
End; {SyntaxSpec.Init}

Procedure SyntaxSpec.CheckElements;

{}Function Max (a,b: MatchType): MatchType;
{}begin
{} if a > b then Max := a else Max := b;
{}End; {Max}

var
    MaxMatch: MatchType; {Holds the highest match found so far}
    SpecsChecked: word;    {Counts the number of specs checked - when silences }
                        {not significant, final count <> length of ElemSpecs.}

begin
    EssPres := TRUE;
    AvgElementScore := 0;
    SpecsChecked := 0;

    {Go through the element spec list, finding which element best matches each }
    {spec, adding this best match score to an accumulated total. }
    ElemSpecs.SetToHead;
    ElemSpecs.SetToNext;
    While not ElemSpecs.AtTail do
    begin
        MaxMatch := 0;

        {For each element ...}
        Elems.SetToHead;
        Elems.SetToNext;
        While not Elems.AtTail do
        begin
            maxMatch := Max (maxMatch,ElemSpecs.Item^.Match(Elems.Item^.e));
            Elems.SetToNext;
        End; {While}

        if (maxMatch = 0) and (ElemSpecs.Item^.Essential)
            then EssPres := FALSE;
        AvgElementScore := AvgElementScore + MaxMatch;
        inc (SpecsChecked);

        ElemSpecs.SetToNext;
    End; {While}

    AvgElementScore := AvgElementScore/SpecsChecked;
End; {SyntaxSpec.CheckElements}

```

```

Procedure SyntaxSpec.CheckSyntax;
var
  EntryPoints: EntryPointList;
  SegMatchSum: word;

begin
  Failures := 0;
  MatchTotal := 0;

  {Loop consists of trying to traverse the syntax tree starting at an }
  {element in the element list. When a failure occurs, try again at the }
  {element after the failed one.}

  Elems.SetToHead;
  Elems.SetToNext;
  While not Elems.AtTail do
  begin
    {get the entry points for the current element}
    EntryPoints.Init (Elems.Item^.e, ElemSpecs);

    if EntryPoints.Empty
    then begin
      Elems.SetToNext;
      inc(Failures);
    end
    else begin
      EntryPoints.SetToEndOfLongestSegment (Elems, SegMatchSum);
      inc(MatchTotal, SegMatchSum);
      if not Elems.AtTail then inc (Failures);
    end;

    EntryPoints.Done (TRUE);
  End; {While}
End; {SyntaxSpec.CheckSyntax}


Destructor SyntaxSpec.Done;
begin
  ElemSpecs.SetToHead;
  ElemSpecs.SetToNext;
  While not ElemSpecs.AtTail do with ElemSpecs.Item^ do
  begin
    Nodes.SetToHead;
    Nodes.SetToNext;
    While not Nodes.AtTail do with Nodes.Item^ do
    begin
      node^.Branches.Done (TRUE);
      dispose (node);
      Nodes.SetToNext;
    End; {While}
    ElemSpecs.SetToNext;
  End; {While}

  ElemSpecs.Done (TRUE);
End; {SyntaxSpec.Done}

{=====END OF OBJECTS==}


VAR
  Candidate: SyntaxSpec;      {Why are these global? In case all their space}
  Elements: ElementList;      {is allocated on the stack }


Procedure CheckSyntax;
type
  elementfiletype = file of elementtype;

var
  Candidates: text;
  ElementFile: elementfiletype;
  CandName: PathStr;
  EssentialsPresent: boolean;
  FailureScore, MatchScore: word;
  PreliminaryScore: real;
  OutString: string[80];
  tempstr: string[5];

```

```

const
  newline = #13#10;
  dashline = '-----'+
    '-----';

begin
  writeln (stdout);

  Elements.Init (ElemFileName); {build an element list from the elements file}

  writeln(stdout,'Recognition scores for call ',UpString(ElemFileName));
  writeln(stdout);
  writeln(stdout, 'CANDIDATE',
    '    ESS  PRELIM    SYN FAILS    SYN SCORE');
  writeln(stdout, '=====',
    '    ==  =====    =====    =====');

  assign(Candidates,CandFileName);
  {$I-} reset(Candidates); {$I+}
  If IOresult <> 0 then
    runerror(UserMessage('Error opening Candidate list file: '+CandFileName));

  While not eof(Candidates) do      {for each syntax spec,}
  begin
    readln(Candidates,CandName);    {read in the name of the spec (.S) file}
    Candidate.Init(CandName);        {build a syntax tree from that file}

    fillchar (OutString[1],80,' ');
    OutString[0] := #80;
    move (Candidate.BirdName[1], OutString[1],length(Candidate.BirdName));

    Candidate.CheckElements (Elements, PreliminaryScore, EssentialsPresent);

    str (PreliminaryScore:45:2, tempstr);
    move (tempstr[1],OutString[45],length(tempstr));

    If EssentialsPresent
    then begin
      OutString[40] := 'Y';
      Candidate.CheckSyntax (Elements, FailureScore, MatchScore);

      str(FailureScore,tempstr);
      move (tempstr[1], OutString[56],length(tempstr));
      str(MatchScore, tempstr);
      move (tempstr[1], OutString[68],length(tempstr));
    end
    else begin
      OutString[40] := 'N';
    end;

    writeln (stdout,OutString);

    Candidate.Done;
  End; {while}

  close (Candidates);
  close (stdout);

  Elements.Done(TRUE);
End (CheckSyntax);

BEGIN
  assign (stdout,'');
  rewrite (stdout);
END.

```

B1.9.2 Full SYNTAX.PAS

```
{
Define MachineResult or TrackResult to get different output from the
default HumanResult. Define Trace to get interactive step-by-step
display of activity. Define Debug in addition for extra debugging during
trace. Default is no Trace or Debug, output to file only.
}

{ $define MachineResult} {Output results in easily machine readable form}
                           {rather than the default human readable form}
{ $define TrackResult}    {Results (human readable only) include failure}
                           {points but exclude preliminary scores}

{ $define Trace} {display each step in the syntax matching process}
{ $define Debug} {Program debugging information output in addition to Trace}

{Leave these defines alone}

{$ifdef Debug}
{$define Trace}
{$endif}

{$define HumanResult}

{$ifdef MachineResult}
{$undef TrackResult}
{$undef HumanResult}
{$endif}

{$ifdef TrackResult}
{$undef MachineResult}
{$undef HumanResult}
{$endif}

{$ifdef MachineResult}
{$undef TrackResult}
{$undef HumanResult}
{$endif}

UNIT SYNTAX;

INTERFACE
USES BCRtypes;

procedure CheckSyntax (ElemFileName, CandFileName: string);
{
  Executes the SYNTAX process

  Identifies an Elements file ('ElemFileName') by comparing it to the
  syntax specs listed in the text file named 'CandFileName'. Output (either
  a table of results (in human or machine readable form) or a detailed
  track of each comparison) goes to standard output. This can be redirected
  to a file, and probably should be if Trace is enabled.

  The machine readable format is similar to the human format - but there are
  no labels or delimiters or text of any kind, and instead of scores being
  left blank when essential elements are not found, a dummy score is inserted.
  For the Match Count, 0 is inserted (since a low match count is "bad") and
  for the Failure Count, 1000 is inserted to simulate a high number of failures.
  The Y/N indication of whether essentials were present is left out. Example:

  Human format
  -----
  <...File Headings....>
  <.....Table headings.....>
  =====
  Bird1      Y      3.7      6      65
  Bird2      N      1.2
  Machine format
  -----
  3.7      6      65
  1.2     1000      0

  The user can feed the output to a plotting package, or some other program that
  extracts the data for its own purposes.
}
```

```

IMPLEMENTATION
USES
  {$ifdef Trace}
  crt, TEXTWIN,
  {$endif}
  {$ifdef TrackResult}
  dos,
  {$endif}
  GENLIB, OBJLIB;

VAR
  stdout: text;    {The initialisation section of this unit assigns}
                   {stdout to '', which give the standard output device}

{$ifdef Trace}
Procedure Report(s: string);
{
  Diagnostic procedure used to inform that an event is about to happen.
}
var
  windw: TextWindow;
begin
  windw.Init (1,1,80,4,Black,White,TRUE,'Report');
  write (s);
  if readkey <> #1 then windw.Done
End;

TYPE
  ScoreObj = object (TextWindow)
    procedure ShowPrelim (p: real);
    procedure ShowSyntax (m,f: word);
  End; {Score}

Procedure ScoreObj.ShowPrelim;
begin
  Activate;
  gotoxy (1,1);
  write ('Prelim: ',p:0:2);
  Deactivate;
End;

Procedure ScoreObj.ShowSyntax;
begin
  Activate;
  gotoxy (1,2);
  write ('Syntax M:',m,' F:',f);
  Deactivate;
End;
{$endif}

TYPE
  IdString = string[30]; {String used to identify element specs and}
                   {syntax tree nodes}
  IdStringPtr = ^IdString;

{=====OBJECT DEFINITIONS==}

{=====SpecFileTextlines OBJECT==}
{
  File object for use with SpecFile (see SyntaxTree, ElemSpecListItem).
  Expected returns a useful error message including the line number at which
  it occurred.
}
{=====}

TYPE
  SpecFileTextLines = object (Textlines)
    function Expected (s: string) :string;
  End;

Function SpecFileTextLines.Expected;
begin
  Expected := 'Error in '+TheFileName+', line '+ Strung(0,0,LinesRead)+
    ': '+s+' expected';
End;

{=====TracedList OBJECT==}
{
  A List with:

  Init - does the normal List.Init, but also a initialises the list's
  window to as specified by the parameters. If the fore- and background
  colours are the same, this is assumed to mean that they should not be
  changed from their current values.
}

```

```

Show - displays the list based on a ShowItem virtual method. The current
item is shown highlighted. The list is shown in a TextWindow specified
by the passed-in 'Win' object.

ShowItem - must do some kind of write (not writeln) using the current item.
}
=====
TYPE
{$ifndef trace}
  TracedList = List;
{$else}
  TracedList = object (List)
    procedure ShowItem; virtual;
    procedure Show (var Win: TextWindow);
  End; {TracedList}

Procedure TracedList.ShowItem;
begin
End; {TracedList.ShowItem}

Procedure TracedList.Show;
var
  CurrentSave: ListItemPtr;
  CurrentSeen: boolean;
  lines: byte;
  lightfore: boolean;
begin
  CurrentSave := Current;
  Win.Activate;
  clrscr;

  lines := 1;
  CurrentSeen := FALSE;
  SetToHead;
  SetToNext;
  While not (AtTail or (CurrentSeen and (lines > Win.d))) do
  begin
    if lines > Win.d
    then begin
      gotoxy(1,1);
      Deline;
      gotoxy(1,Win.d);
    end
    else gotoxy (1,lines);
    if Current = CurrentSave
    then begin
      lightfore := Win.Fcol in [White];
      textbackground (White*byte(lightfore));
      textcolor (White*byte(not lightfore));
      CurrentSeen := TRUE;
      ShowItem;
      textcolor(Win.Fcol);
      textbackground(Win.Bcol);
    end
    else ShowItem;
    inc (lines);
    SetToNext;
  End;

  Win.Deactivate;
  Current := CurrentSave;
End; {TracedList.Show}

{$endif}

=====ElementListItem OBJECT==
{
  Item object for ElementList. The Init loads an elementtype into the 'e'
  field
}
=====
TYPE

ElementListItem = Object (ListItem)
  e: elementtype;

  constructor Init (elem: elementtype); {create list from element file}
  {$ifdef Debug}
  destructor Done; virtual;
  {$endif}
End; {ElementListItem}

ElemListItemPtr = ^ElementListItem;

```

```

Constructor ElementListItem.Init;
begin
  e := elem;
End; {ElementListItem}

{$ifdef Debug}
Destructor ElementListItem.Done; {testing purposes}
begin
  Report ('Destroying element: '+ClassName[e.class]+' '+Strung(5,2,e.durn));
End;
{$endif}

{=====ElementList OBJECT==}
{
  List of elements as read in from .E file. Item returns a pointer to the
  current ElemListItem object, without detaching it from the list.
  Init reads in elements from .E file.

  CopyInit is meant to be used for lists that dont actually have their own
  list structure associated with them, but will be used as handles on an
  existing list structure, i.e. they will be assigned to another ElementList.
  For some reason the assignment has no effect if the handle has not been
  initialised, even though no use of virtual methods are implied in the
  process. Strange. When tracing is on, CopyInit creates a separate window
  for the copied list. CopyDone removes the separate window if tracing is on
  but does nothing otherwise.
}
{=====}
TYPE

ElementList = Object (TracedList)
{$ifdef Trace}
  procedure ShowItem; virtual;
{$endif}

  constructor Init (ElemFileName: string);
  constructor CopyInit (Source: ElementList);
  function Item: ElemListItemPtr;
End; {ElementList}

Constructor ElementList.CopyInit;
begin
  Self := Source;
End;

Constructor ElementList.Init;
var
  efile: file of elementtype;
  e: elementtype;
  i: ElemListItemPtr;
begin
  TracedList.Init;

  assign (efile, ElemFileName);
  {$I-} reset (efile); {$I+ }
  if IOresult = 0
  then begin
    While not eof(efile) do
    begin
      read (efile,e);
      Attach (new(ElemListItemPtr,Init(e)));
    end; {While}
    close (efile);
  end
  else runerror(UserMessage ('Error opening element file '+ElemFileName));
End; {ElementList.Init}

function ElementList.Item;
begin
  Item := ElemListItemPtr(List.Item(FALSE));
End; {ElementList.Item}

{$ifdef Trace}
procedure ElementList.ShowItem;
begin
  write (Classname[Item^.e.Class], ' ',Item^.e.durn:0:2);
End; {ElementList.ShowItem}
{$endif}

```

```

TYPE
{
  Because we require delayed compiler-binding of pointer type identifiers,
  we had to have everything under one type definition, which meant
  separating the methods from the type declarations for the following objects:
  SynTreeNodeList/Item, ElemSpecList/Item and SyntaxTree. Sorry.
}

  SynTreeNodePtr = ^SynTreeNode;

{-----SynTreeNodeListItem OBJECT--}
{
  Item object for SynTreeNodeList type. Init accepts a pointer to a
  SyntaxTree Node.
}
{-----}
  SynTreeNodeListItem = object (ListItem)
    node: SynTreeNodePtr;
    constructor Init (p: SynTreeNodePtr);
    {$ifdef Debug}
    destructor Done; virtual;
    {$endif}
  End; {SynTreeNodeListItem}

  SynTreeNodeListItemPtr = ^SynTreeNodeListItem;

{-----SynTreeNodeList OBJECT--}
{
  A list of pointers to the nodes in a syntax tree. Used for the 'Nodes' list
  in the ElemSpec object and the 'Branches' list in the SynTreeNode record.
  Item returns a pointer to the current SyntaxTree node, without detaching.
  Append allows another SynTreeNodeList to be inserted into the list at the
  current position, without changing that position.
}
{-----}
  SynTreeNodeList = object (TracedList)
    function Item: SynTreeNodeListItemPtr;
    procedure Append (STNList: SynTreeNodeList);

    {$ifdef Trace}
    procedure ShowItem; virtual;
    {$endif}
  End; {SynTreeNodeList}

{-----ElemSpecListItem OBJECT--}
{
  Item object for ElemSpecList.
  Init reads in an element spec from a file 'f' which must be open and
  positioned at the start (after the label) of an element spec as defined in
  a .E file. The identifier of the spec is passed in 'name', and whether it
  is essential or not, in 'essentialflag'.
  Match compares the current item with a passed in element 'element',
  returning 0 if the durations do not match, 1 if the durations match,
  2 if the frequencies match as well, 3 if the class occurrences match as well.
  Silent returns true if the spec is for a silent element.
}
{-----}
  MatchType = 0..4;

  ElemSpecListItem = object (ListItem)
    identifier: IdString;
    Essential: boolean;
    freqmin, freqmax: real;
    durnmin, durnmax: real;
    profile: array[single..noise] of byte;
    Nodes: SynTreeNodeList;
    OccurrenceDistance: real;

    constructor Init (var sf:SpecFileTextLines; name: string;
      essentialflag: boolean);
    function Match (element: elementtype): MatchType;
    function SilentSpec: boolean;
    destructor Done; virtual;
  End; {ElemSpecListItem}

  ElemSpecListItemPtr = ^ElemSpecListItem;

{-----ElemSpecList OBJECT--}
{
  List of element specifications - to be matched against the input elements -
  as read in from the Element Specification part of the .S file. This object
  is only used inside a SyntaxSpec. Item returns a pointer to the current
  ElemSpecListItem, without detaching.
}

```

```

{-----}
ElemSpecList = object (TracedList)
  function Item: ElemSpecListItemPtr;
End; {ElemSpecList}

{-----SyntaxSpec OBJECT-----}
{
A list ('ElemSpecs') and a tree. The tree structure's nodes are SynTreeNode,
which each contain a pointer ('ElemSpec') to an item in the ElemSpecs list,
and a list 'SynTreeNodeList' of pointers to other tree nodes.
The two main methods, which take an ElemList object 'Elems' as their
input, are:

PreliminaryScore - scores the call by averaging the scores of the matches
between the 'Elems' items and the 'ElemSpec' items over the number of
'Elems' items. 'EssPres' id true if all specs marked essential match an
element in 'Elems'.

CheckSyntax - scores the call by traversing the syntax tree based on the
input 'Elems' (ie comparing each element in turn with the 'ElemSpec'
at the appropriate node and counting the discontinuities in the traversal,
which is returned in FailureScore. 'MatchScore' is the sum of the match
levels obtained for each element in the call.

The Init method creates the ElemSpecs list by reading the headers in the
ELEMENTS: part of the .S file (whose name is passed to Init as
'CandidateName') and then building ElemSpecListItems, passing them the
correct file position in their Init. It builds the syntax tree by reading
the information under the SYNTAX: part of the .S file. It also reads the
'BirdName' from the .S file.

The Done method disposes all the SynTreeNodes and the ElemSpecs list.
}
{-----}
SynTreeNode = record
  Identifier: IdString;
  ElemSpec: ElemSpecListItemPtr;
  Branches: SynTreeNodeList;
end;

PathStr = string[79];
SyntaxSpec = object
  BirdName: string[40];
  ElemSpecs: ElemSpecList;

  constructor Init (CandidateName: PathStr);
  procedure CheckElements (Elems: ElementList;
    var AvgElementScore: real;
    var EssPres: boolean);
  procedure CheckSyntax (Elems: ElementList;
    var Failures, MatchTotal: word);
  destructor Done;
End; {SyntaxSpec}

{=====SegmentTrackItem OBJECT=====}
{
Item object for the TrackSegment list object. Records a node that matched
an element, and the match score.
}
{=====}
TYPE
  SegmentTrackItem = Object (ListItem)
    NodeId: IdString;
    Score: MatchType;

    constructor Init (N: IdString; S: MatchType);
  End; {SegmentTrackItem}

  SegmentTrackItemPtr = ^SegmentTrackItem;

Constructor SegmentTrackItem.Init;
begin
  ListItem.Init;
  NodeId := N;
  Score := S;
End;

```

```

{$ifdef TrackResult}
{=====SegmentTrack OBJECT==}
{
  Object used to track the way a segment of the incoming call matches the
  syntax tree. It stores the matching nodes' names and their scores.
  'Output' lists the matching node names, against the identifiers of the
  elements in the passed in Seg. ('Seg is the element list, 'StartSeg' points
  to the first item in the segment). Output is sent to the open text file
  passed in ('TrackFile').
}
{=====}

TYPE
  SegmentTrack = Object (SynTreeNodeList)
    function Item: SegmentTrackItemPtr;
    procedure Output (Seg: ElementList; StartSeg: ElemListItemPtr;
      var TrackFile: text);
  End; {SegmentTrack}

Function SegmentTrack.Item;
begin
  Item := SegmentTrackItemPtr (List.Item(FALSE));
End;

Procedure SegmentTrack.Output;
begin
  {find the start of the segment}
  Seg.SetToHead;
  Seg.SetToNext;
  while not Seg.AtTail and (StartSeg <> Seg.Item) do Seg.SetToNext;
  if Seg.AtTail then
    runerror(UserMessage('Start of segment to trace not found'));

  {go through the Tracer list outputting stuff to file}
  SetToHead;
  SetToNext;
  While not AtTail do
  begin
    with Seg.Item^.e do write(TrackFile,'| ', classname[class],' ',durn:4:1,' --- ');
    with Item^ do writeln (TrackFile, NodeId,' ',Score);
    SetToNext;
    Seg.SetToNext;
  End; {While}
End;
{$endif}

{=====EntryPointList OBJECT==}
{
  List of nodes (entry points) in the tree that match a particular element.
  Init finds the entry points for the passed in 'Element', by searching
  through 'Specs' for a match. This is done by appending the Nodes list of
  any Specs item that matches the current element.

  SetToLongestSegment is where the comparison between the elements and the
  syntax tree actually takes place. It sets the current position of the
  passed in ElementList, 'ReturnElementst', to the end (actually one after the
  end) of the longest continuous (unbroken by failures to match) segment of
  elements, based on the initial current position of 'ReturnElements', the
  list of entry points and the syntax tree structure. Also returns the score
  (sum of match levels ) of the segment in 'SegmentScore'.
}
{=====}

TYPE
  EntryPointList = object (SynTreeNodeList)
    {$ifdef Trace}
    procedure ShowItem; virtual;
    {$endif}

    constructor Init (EntryElement: elementtype; Specs: ElemSpecList);
    procedure SetToEndOfLongestSegment
      (var ReturnElements: ElementList;
      var SegmentScore: word);
  End; {EntryPointList}

{$ifdef Trace}
Procedure EntryPointList.ShowItem;
begin
  with Item^.node^ do
  begin
    write (Identifier,' (');
    if ElemSpec <> NIL then write (ElemSpec^.Identifier) else write ('Label');
    write (')');
  end;
end;

```

```

end;
End; {EntryPointList.ShowItem}
{$endif}

Constructor EntryPointList.Init;
begin
  TracedList.Init;

  Specs.SetToHead;
  Specs.SetToNext;
  While not Specs.AtTail do with Specs.Item^ do
    begin
      if Match (EntryElement) > 0 then Append (Nodes);
      Specs.SetToNext;
    end; {While}
  End; {EntryPointList.Init}

{$ifdef Trace}
VAR
  EntryWin: TextWindow; {Window for putting the current entrypoinits list in.}
                        {must be accessible to Checksyntax too, so global. }
{$endif}

Procedure EntryPointList.SetToEndOfLongestSegment;
var
  Segment, LongestSegment: ElementList;
  SegmentLength, MaxSegmentLength: word;
  ThisSegmentScore, MaxSegmentScore: word;

{$ifdef Trace}
SegWin: TextWindow;      {Shows the element list being traversed.    }
{$endif}

{$ifdef TrackResult}
CurrentTrack, LongestTrack, SpareTrack: ^SegmentTrack;
  {point to the SegmentTrack objects being used }
{$endif}

()Procedure FindEndOfSegment (SyntaxTreeNode: SynTreeNodePtr);
()
() Tries to follow the syntax tree using the element sequence in 'Segment'.
() The latter's current position is left at the element which first failed to
() match a branch node. The count of elements which succesfully matched nodes
() is placed in 'SegmentLength', and the sum of the match levels of the
() segment's elements is placed in 'SegmentScore'. 'SyntaxTreeNode' is the
() node in the syntax tree where parsing starts, and is assumed to match the
() initial current element in 'Segment'.
()
()var
() { $ifdef Debug}
()   reportstr: string [80];
() { $endif}
()
() { $ifdef Trace}
()   MatchWin,           {Shows the match between the current branch and}
()                       {the current segment element}
()   AllWin: TextWindow; {Shows the current node and its branches.}
() { $endif}
()
()   MatchLevel, MaxMatch: MatchType;
()   Branch, MaxBranch: SynTreeNodePtr;
()   Element: elementtype;
()   PastLabels,           {Used to check for infinite loops in labels}
()   AllBranches: SynTreeNodeList; {Local copy of the node's branch list with}
()                               {any label node's branch lists appended }
()
()()Function NodeSeen (STNList: SynTreeNodeList):boolean;
()() Checks STNList to see if the node of the item that was current when the
()() function was called appears previously in the list, returning TRUE if so.
()()var
()() tempBAE: boolean;
()() CheckNode: SynTreeNodeListItemPtr;
()()begin
()() tempBAE := FALSE;
()() CheckNode := STNList.Item;
()() STNList.SetToHead;
()() STNList.SetToNext;
()() While not ((STNList.Item=CheckNode) or tempBAE
()()           or STNList.AtTail {for safety - shouldn't happen}) do
()()   begin
()()     tempBAE := (CheckNode^.node = STNList.Item^.node);
()()     STNList.SetToNext;
()()   end; {While}

```

```

({}) NodeSeen := tempBAE;
({})End; {NodeIn}
({})
({})begin
({}) {Handle the first element in the segment separately since the loop}
({}) {checks the BRANCHES of the current syntax tree node.}
({})
({}) SegmentLength := 1; {The first element is known to match}
({}) ThisSegmentScore := SyntaxTreeNode^.ElemSpec^.Match (Segment.Item^.e);
({})
({}) {To save this match from being redone you could store a matchlevel in }
({}) {in the entriypointlist item and pass it to this sub-proc, but it ain't }
({}) {worth the effort}
({})
({}) MaxMatch := ThisSegmentScore;
({}) Segment.SetToNext;
({})
({}) {$ifdef TrackResult}
({}) {record the node name and score in the segment tracker}
({}) CurrentTrack^.Attach (new (SegmentTrackItemPtr,
({}) Init (SyntaxTreeNode^.Identifier, MaxMatch)));
({}) {$endif}
({})
({}) {Go through the segment and the tree, checking if elements match a branch}
({}) { off nodes}
({}) While (not Segment.AtTail) and (MaxMatch > 0) do
({}) begin
({}) {$ifdef Debug}
({}) reportstr := 'About to destroy the allbranches list for tree node: ' +
({}) SyntaxTreeNode^.Identifier;
({}) {$endif}
({})
({}) {$ifdef Trace}
({}) Allwin.Init (19,18,43,7,Yellow,Blue,TRUE,SyntaxTreeNode^.Identifier);
({}) {$endif}
({})
({}) Element := Segment.Item^.e;
({}) MaxMatch := 0;
({}) AllBranches.Init;
({}) AllBranches.Append (SyntaxTreeNode^.Branches);
({}) AllBranches.SetToHead;
({}) AllBranches.SetToNext;
({})
({}) {$ifdef Trace}
({}) Segment.Show (SegWin);
({}) {$endif}
({})
({}) {Go through the syntax tree node's branch list looking for the branch}
({}) {that best matches the current Segment element}
({}) While not AllBranches.AtTail do
({}) begin
({}) {$ifdef Trace}
({}) AllBranches.Show(AllWin);
({}) Report ('About to compare a branch with a segment');
({}) {$endif}
({})
({}) Branch := AllBranches.Item^.node;
({}) if Branch^.ElemSpec = NIL
({}) then begin
({}) {This is a label node so add its branches to AllBranches}
({}) {But first make sure we haven't already had this label node -}
({}) {in which case we are going round in circles}
({})
({}) if not NodeSeen(AllBranches)
({}) then AllBranches.Append (Branch^.Branches)
({}) else runerror (UserMessage('Self referential label node path encountered'));
({}) end
({}) else begin {A normal branch node}
({}) MatchLevel := Branch^.ElemSpec^.Match (Element);
({}) {$ifdef Trace}
({}) if MatchLevel <> 0
({}) then begin
({}) MatchWin.Init (57,20,10,3,White,Green,TRUE,'Match');
({}) write (matchlevel,' Max:', MaxMatch);
({}) end
({}) else begin
({}) MatchWin.Init (57,20,10,3,Black,Red,TRUE,'Match');
({}) write ('FAILED');
({}) end;
({}) Report ('Result of match is being displayed');
({}) MatchWin.Done;
({}) {$endif}
({})
({}) if MatchLevel > MaxMatch
({}) then begin
({}) MaxMatch := MatchLevel;

```

```

{}          MaxBranch := Branch;
{}          end;
{}          end;
{}          AllBranches.SetToNext;
{}          End; {While}
{}
{}          {if a matching branch was found, advance the element and the node}
{}          If MaxMatch > 0 then
{}          begin
{}              inc (SegmentLength);
{}              inc (ThisSegmentScore,MaxMatch);
{}              Segment.SetToNext;
{}              SyntaxTreeNode := MaxBranch;
{}
{}              {$ifdef TrackResult}
{}              {record the node name and score in the segment tracker}
{}              CurrentTrack^.Attach (new (SegmentTrackItemPtr,
{}                  Init (SyntaxTreeNode^.Identifier, MaxMatch)));
{}              {$endif}
{}          End;
{}
{}          {$ifdef Debug}
{}          Report (reportstr);
{}          {$endif}
{}
{}          {$ifdef Trace}
{}          AllWin.Done;
{}          {$endif}
{}
{}          AllBranches.Done (TRUE); {Destroy the list items}
{}      End; {While}
{}End; {FindEndOfSegment}

{$ifdef Trace}
var
    LongWin: TextWindow;
{$endif}

begin
    {$ifdef Trace}
    LongWin.Init (39,5,23,4,Yellow,Cyan,TRUE,'Longest Segment');
    Longwin.Deactivate;
    SegWin.Init (62,5,18,20,Yellow,Blue,TRUE,'Segment');
    SegWin.Deactivate;
    {$endif}

    {$ifdef TrackResult}
    new (LongestTrack, Init);
    {$endif}

    MaxSegmentLength := 0;
    SetToHead;
    SetToNext;
    While not AtTail do
    begin
        {The segment starts off identical to element list}
        Segment.CopyInit (ReturnElements);

        {$ifdef Trace}
        Show(EntryWin);
        Segment.Show(SegWin);
        {$endif}

        {$ifdef TrackResult}
        new (CurrentTrack, Init);
        {$endif}

        {Now return Segment with current position pointing after the end,}
        {together with SegmentLength and SegmentScore}
        FindEndOfSegment (Item^.node);

        {$ifdef Trace}
        if not Segment.AtTail then with Segment.Item^.e do
        Report ('Entry Point: '+Item^.node^.Identifier+' Length: '+
            Strung(0,0,SegmentLength)+' Failed at: '+Classname[class]+' '+
            Strung(0,2,durn));
        {$endif}

        {See if this is the longest segment so far, and preserve it if so}
        If SegmentLength > MaxSegmentLength then
        begin
            SegmentScore := ThisSegmentScore;
            MaxSegmentLength := SegmentLength;
            LongestSegment.CopyInit(Segment);

```

```

    {$ifdef TrackResult}
    {swap longest and current tracks}
    SpareTrack := LongestTrack;
    LongestTrack := CurrentTrack;
    CurrentTrack := SpareTrack;
    {$endif}
End; {If}

{$ifdef Trace}
LongestSegment.Show(LongWin);
LongWin.Activate;
gotoxy (1,1);
write ('Length: ',MaxSegmentLength,' ');
LongWin.Deactivate;
{$endif}

{$ifdef TrackResult}
dispose (CurrentTrack, Done(TRUE));
{$endif}

SetToNext;
End; {While}

{$ifdef TrackResult}
{Output the track to file - note ReturnElements still has the start of }
{ the segment as its current item }
LongestTrack^.Output (LongestSegment, ReturnElements.Item, stdout);
dispose (LongestTrack, Done(TRUE));
{$endif}

{Set the element list so that the current position points after the end of}
{the longest segment. Return the score.}
ReturnElements := LongestSegment;

{$ifdef Trace}
SegWin.Done;
Report ('Longest segment found. Length: '+
        Strung(0,0,MaxSegmentLength)+' elements');
LongWin.Done;
{$endif}

End; {EntryPoints.SetToEndOfLongestSegment}

{-----SynTreeNodeListItem METHODS--}
Constructor SynTreeNodeListItem.Init;
begin
    node := p;
End; {SynTreeNodeListItem.Init}

{$ifdef Debug}
Destructor SynTreeNodeListItem.Done;
begin
    Report ('Destroying SynTreeNodeListItem item pointing to node named '+
            node^.Identifier);
End;
{$endif}

{-----SynTreeNodeList METHODS--}
Function SynTreeNodeList.Item;
begin
    Item := SynTreeNodeListItemPtr(List.Item (FALSE));
End; {SynTreeNodeList.Item}

Procedure SynTreeNodeList.Append;
var
    CurrentSave: ListItemPtr;
begin
    CurrentSave := Current;           {We don't want to change the current pos}
    STNList.SetToHead;                {Note STNList is a local copy so this }
    STNList.SetToNext;                {won't affect the state of the original. }
    While not STNList.AtTail do
    begin
        Attach(new(SynTreeNodeListItemPtr, Init (STNList.Item^.node)));
        STNList.SetToNext;
    end; {While}
    Current := CurrentSave;
End; {SynTreeNodeList.Append}

```

```

{$ifdef Trace}
Procedure SynTreeNodeList.ShowItem;
begin
  With Item^.node^ do begin
    write(Identifier);
    if ElemSpec <> NIL then write (' (' ,ElemSpec.Identifier,')');
  End;
End;
{$endif}

(-----ElemSpecListItem METHODS--)
Constructor ElemSpecListItem.Init;
var
  line: string;
  seq: StringSeq;
  error: integer;

()procedure varset (var var1, var2: real; varlabel: string);
()begin
()  if UpString(seq.Getword) <> varlabel then
()    runerror (UserMessage(sf.Expected('"' + varlabel + '"')));
()  val (seq.Getword, var1, error);
()  if error <> 0 then
()    runerror (UserMessage(sf.Expected(varlabel + 'number')));
()  if (UpString(seq.Getword) <> 'TO') then
()    runerror (UserMessage(sf.Expected('TO')));
()  val (seq.Getword, var2, error);
()  if error <> 0 then
()    runerror (UserMessage(sf.Expected(varlabel + 'number')));
()end;

var
  i: classtype;
begin
  seq.Init(sf.NextLine);
  if Upstring(seq.Getword) = 'SILENCE'
  then begin
    {fillchar (Self,SizeOf(Self), #0); messed up the object - possibly
    the VMT address got trashed - anyway got error 210 Object not
    initialised when tried executing Done.}
    freqmin := 0;
    freqmax := 0;
    seq.Reset;
    varset (durnmin, durnmax, 'SILENCE');
  end
  else begin
    {Set the spec fields}
    seq.Reset;
    varset (freqmin, freqmax, 'FREQUENCY');
    seq.Init(sf.NextLine);
    varset (durnmin, durnmax, 'DURATION');
    seq.Init (sf.NextLine);
    i := single;
    repeat
      if UpString (seq.GetWord) <> classname[i] then
        runerror (UserMessage(sf.Expected(classname[i])));
      val (seq.Getword, profile[i], error);
      if error <> 0 then runerror(UserMessage(sf.Expected('Number')));
      inc(i);
    until (i > noise);
    seq.Init(sf.NextLine);
    if UpString(seq.Getword) <> 'OCCURRENCE DISTANCE' then
      runerror(UserMessage(sf.Expected('OCCURRENCE_DISTANCE')));
    val (Seq.GetWord, OccurrenceDistance,error);
    if error <> 0 then runerror(UserMessage(sf.Expected('Number')));
  end;
  identifier := name;
  Essential := essentialflag;

  {Create an empty nodes list - this will be expanded by someone else}
  Nodes.Init;
End; {ElemSpecListItem.Init}

Destructor ElemSpecListItem.Done;
begin
  {$ifdef Debug}
  Report('Destroying element spec called '+identifier+
  ' - about to destroy its node list');
  {$endif}

  Nodes.Done (TRUE);
End; {ElemSpecListItem.Done}

```

```

Function ElemSpecListItem.SilentSpec;
begin
  SilentSpec := (freqmax = 0) and (freqmin = 0);
  {This is just how it currently signals a silence - it would be }
  {computationally more efficient to have an additional silence or }
  {class flag in the ElemSpecListItem, and to set this flag when }
  {reading in the soec from the .S file. }
End;

Function ElemSpecListItem.Match;
var
  i: classtype;
  sumsqr: real;
begin
  Match := 0;
  if element.class = silent
  then begin
    if ((durnmin = 0) and (durnmax = 0)) {durnmin=durnmax=0 signals that}
    {the spec matches any silence }
    then Match := 4
    else if SilentSpec {Full score if the right}
    {length, otherwise minimum score}
    then if (durnmin <= element.durn) and (element.durn <= durnmax)
    then Match := 4 else Match := 1;
  end
  else if not SilentSpec and
  (durnmin <= element.durn) and (element.durn <= durnmax) then
  begin
    Match := 2; {durations match}
    If (element.freq >= freqmin) and (element.freq <= freqmax) then
    begin
      Match := 3; {frequencies match too}

      {Check distance between class occurrence vector }
      { and spec profile vector and give 4 if the distance is small }
      sumsqr := 0;
      for i := single to noise do
      sumsqr := sumsqr+sqr(profile[i]-100*element.occurrence[i]);
      if sqrt(sumsqr) <= OccurrenceDistance then Match := 4;
    end; {If}
  end {If};
End {ElemSpecListItem.Match};

{-----ElemSpecList METHODS--}
Function ElemSpecList.Item;
begin
  Item := ElemSpecListItemPtr(List.Item(FALSE));
End; {ElemSpecList.Item}

{-----SyntaxSpec METHODS--}
Constructor SyntaxSpec.Init;
var
  SpecFile: SpecFileTextLines;
  firstword: string[40];
  seq: StringSeq;
  name: IdString;
  essential: boolean;

{}Procedure SkipToHeading;
{}
{} puts the first word of the next non-blank line in 'firstword', and the
{} entire line is in 'seq' with the current postion pointing at the second
{} word. 'firstword' is converted to uppercase.
{}
{}begin
{}  repeat
{}    seq.Init(SpecFile.NextLine);
{}    firstword := UpString(Seq.GetWord);
{}  until firstword <> '';
{}End; {SkipToHeading}

{}Function FindHeading (heading:string): string;
{}
{} Returns the remainder of the line in 'SpecFile' that starts with 'heading'
{} or halts with run time error if that line is not found.
{}
{}begin
{} {SpecFile.Open ('%%Reopen%%'); here to allow headings in any order}
{} {in the .S file}
{}  repeat
{}    SkipToHeading
{}  until (firstword = UpString(heading)) or (firstword = '%%END');
{}  if firstword = '%%END' then

```

```

{}      runerror (UserMessage(SpecFile.Expected(heading)));
{}      FindHeading := seq.Rest;
{}End; {FindHeading}

{}Procedure CreateElemSpecs;
{}
{}  () Read element specs from the .S file and build the ElemSpecs list.
{}  () Stops when "SYNTAX:" is reached.
{}
{}begin
{}  ElemSpecs.Init;
{}
{}  SkipToHeading;
{}  {we have reached an element spec label, so check if its essential      }
{}  {and then make an item, attach it, skip blanks to the next label until }
{}  {the end of ELEMENTS: , ie SYNTAX: is reached.                        }
{}
{}  While not ((firstword = '%%END') or (firstword = 'SYNTAX:')) do
{}  begin
{}    name := firstword;
{}    if name[length(name)] = ':'
{}    then essential := (UpString(seq.GetWord) = 'ESSENTIAL')
{}    else runerror (UserMessage(SpecFile.Expected('Colon ":"')));
{}
{}    {Create the spec object and add it to the end of the list}
{}    ElemSpecs.Attach(new(ElemSpecListItemPtr, Init(SpecFile, name, essential)));
{}
{}    SkipToHeading;
{}  End; {While}
{}  if firstword = '%%END' then
{}  runerror (UserMessage(SpecFile.Expected('SYNTAX:')));
{}End; {CreateElemSpecs}

{}Procedure CreateSyntaxTree;
{}
{}  () Read node specs from the .S file and build the syntax tree.
{}  () Stops when the end of the file is reached.
{}
{}var
{}  TempNodeList: SynTreeNodeList; {holds the nodes until they can be linked}
{}                                  {correctly on a second pass}
{}  AnotherNodeList: SynTreeNodeList; {Another copy of the TempNodeList}
{}  TempId: IdStringPtr;
{}
{}begin
{}  TempNodeList.Init;
{}
{}  { Remember node spec format is                                     }
{}  { <node identifier>:                                             }
{}  { <element spec name> <list of branch node identifiers> }
{}
{}  {FIRST PASS: READ}
{}
{}  {Go through file adding nodes to the temporary list. Force the Branches to}
{}  {point to strings containing node names rather than the nodes themselves, }
{}  {which may not have been read in yet.}
{}
{}  SkipToHeading; {Look for first node}
{}  While not (firstword = '%%END') do
{}  begin
{}    {Get node identifier}
{}    name := firstword;
{}    if name[length(name)] <> ':'
{}    then runerror (UserMessage(SpecFile.Expected('Colon ":"')));
{}
{}    {Make a new node, attach it and give it its name}
{}    TempNodeList.Attach (new(SynTreeNodeListItemPtr, Init(new(SynTreeNodePtr))));
{}    TempNodeList.Item^.node^.Identifier := name;
{}
{}    {Get node details...}
{}    seq.Init (SpecFile.NextLine);
{}    if seq.EndOfSequence
{}    then runerror (UserMessage(SpecFile.Expected('Element spec identifier')));
{}
{}    {... node element spec name...}
{}    name := UpString(seq.GetWord)+'.';
{}
{}    if name <> '..'
{}    then begin {A normal node}
{}      {Search for the element with that identifier in ElemSpecs}
{}      ElemSpecs.SetToHead;
{}      ElemSpecs.SetToNext;
{}      while (not ElemSpecs.AtTail) and (ElemSpecs.Item^.Identifier <> name)
{}      do ElemSpecs.SetToNext;
{}      if ElemSpecs.AtTail then
{}      runerror (UserMessage ('Unknown element spec identifier in line '+

```

```

({} Strung(0,0,SpecFile.LinesRead)+' of '+SpecFile.TheFileName));
({}
({} {Set node's element spec pointer to the found element spec...}
({} TempNodeList.Item^.node^.ElemSpec := ElemSpecs.Item;
({}
({} {... and install the complementary pointer, ie:
({} {Add a pointer to the node onto the found element spec's 'Nodes' list}
({} ElemSpecs.Item^.Nodes.Attach(new(SynTreeNodeListItemPtr,
({} Init(TempNodeList.Item^.node)));
({} end
({} else TempNodeList.Item^.node^.ElemSpec := NIL; {A label node}
({}
({} {... initialise the node branch list. Get branch node names and make }
({} {the branches point to the names. The names will be destroyed on the }
({} {second pass, ie when the branches are set to their correct places. }
({} TempNodeList.Item^.node^.Branches.Init;
({} While not seq.EndOfSequence do
({} begin
({} new(TempId);
({} TempId^ := UpString(seq.GetWord);
({} TempNodeList.Item^.node^.Branches.Attach
({} (new (SynTreeNodeListItemPtr, Init(SynTreeNodePtr(TempId))));
({} End; {While}
({}
({} {Go to next node spec}
({} SkipToHeading;
({} End; {While}
({}
({} {SECOND PASS: RESOLVE}
({}
({} AnotherNodeList.Init;
({} AnotherNodeList := TempNodeList; {Another is a doppelganger - even it's own
({} {Head and Tail aren't used - so it mustn't}
({} {Be Done'd - else Temp can't be.}
({}
({} {Go through the temp node list resolving syntax tree node references}
({} {in the nodes' branch lists. }
({} TempNodeList.SetToHead;
({} TempNodeList.SetToNext;
({} While not TempNodeList.AtTail do With TempNodeList.Item^.node^ do
({} begin
({} {$ifdef Debug}
({} if ElemSpec <> NIL
({} then Report ('Resolving node branches: '+Identifier+
({} ' ('+ElemSpec^.Identifier+')')
({} else Report ('Resolving node branches: '+Identifier+' (NO ELEMENT SPEC)');
({} {$endif}
({}
({} {Go through the branches of the current node, matching identifiers with}
({} {the names in the NodeList (use AnotherNodeList because otherwise you }
({} {mess up the current position of TempNodeList. }
({} Branches.SetToHead;
({} Branches.SetToNext;
({} While not Branches.AtTail do
({} begin
({} TempId := IdStringPtr(Branches.Item^.node);
({}
({} {$ifdef Debug}
({} Report ('Looking for branch node '+TempId^);
({} {$endif}
({}
({} {Go through the node list looking for nodes with the name pointed to}
({} {by the current branch. }
({} AnotherNodeList.SetToHead;
({} AnotherNodeList.SetToNext;
({} while (not AnotherNodeList.AtTail) and
({} (TempId^+':' <> AnotherNodeList.Item^.node^.Identifier) do
({} AnotherNodeList.SetToNext;
({} if AnotherNodeList.AtTail then
({} runerror (UserMessage ('Unknown node identifier: '+TempId^+
({} ' in branch list of '+ElemSpec^.Identifier));
({}
({} {Found the identifier, so make the branch point to that node and }
({} {destroy the name it had been pointing to. }
({} {Note that in order to do this efficiently, it was necessary to }
({} {write to the item objects 'node' field - otherwise we would have }
({} {had to create a new item object, attach it into the list and }
({} {detach the old one - a hassle. }
({} Branches.Item^.node := AnotherNodeList.Item^.node;
({} dispose(TempId);
({}
({} Branches.SetToNext;
({} End; {While}
({}
({} TempNodeList.SetToNext;
({} End; {While}

```

```

{}
{} {Destroy the list - the SynTreeNode's are not destroyed because that}
{} {would have to be coded into SynTreeNodeListItem.Done, which I have}
{} {thoughtfully refrained from doing. The tree of SynTreeNode's is a }
{} {separate to any list of SynTreeNode's that might be used to reference}
{} {it, and so should be destroyed by SynTaxTree.Done directly and not}
{} {by any SynTreeNodeListItem.Done}
{}
{} {$ifdef Debug}
{} Report ('About to destroy the temporary list of nodes of the newly-constructed tree');
{} {$endif}
{}
{} TempNodeList.Done (TRUE);
{}End; {CreateSyntaxTree}

begin {Init}

  {Open the .S file}
  {$I-} SpecFile.Open (CandidateName+'.S'); {$I+}
  if IOresult <> 0 then
    runerror (UserMessage('Error opening candidate file: '+CandidateName+'.S'));

  {Find the bird name}
  BirdName := FindHeading('BIRD:');

  {Find the "ELEMENTS:" heading}
  if FindHeading ('ELEMENTS:') = '' then; {Ignore what's after ELEMENTS, just }
                                         {place the file position}

  {Create the ElemSpecs list from the information under ELEMENTS;}
  CreateElemSpecs;

  {Create syntax tree from information under SYNTAX;}
  CreateSyntaxTree;

End; {SyntaxSpec.Init}

{$ifdef Trace}
VAR
  Score: ScoreObj;           {Must be seen by SyntaxSpec procedures as well}
                             {as main procedure}
{$endif}

Procedure SyntaxSpec.CheckElements;

{}Function Max (a,b: MatchType): MatchType;
{}begin
{}  if a > b then Max := a else Max := b;
{}End; {Max}

var
  MaxMatch: MatchType; {Holds the highest match found so far}
  SpecsChecked: word;   {Counts the number of specs checked - when silences }
                       {not significant, final count <> length of ElemSpecs.}

begin
  EssPres := TRUE;
  AvgElementScore := 0;
  SpecsChecked := 0;

  {Go through the element spec list, finding which element best matches each }
  {spec, adding this best match score to an accumulated total.}
  ElemSpecs.SetToHead;
  ElemSpecs.SetToNext;
  While not ElemSpecs.AtTail do
    begin
      {$ifdef Debug}
      Report('Checking preliminary score for '+ElemSpecs.Item^.Identifier);
      {$endif}

      MaxMatch := 0;

      {For each element ...}
      Elems.SetToHead;
      Elems.SetToNext;
      While not Elems.AtTail do
        begin
          maxMatch := Max (maxMatch,ElemSpecs.Item^.Match(Elems.Item^.e));

          {$ifdef Debug}
          with Elems.Item^.e do
            Report ('Checked against: '+Classname[class]+' '+ Strung (4,1,durn)
              +' highest match so far: '+Strung (0,0,MaxMatch));
          {$endif}

          Elems.SetToNext;
        end;
      {While}
    end;
  end;

```

```

    if (maxMatch = 0) and (ElemSpecs.Item^.Essential)
    then EssPres := FALSE;
    AvgElementScore := AvgElementScore + MaxMatch;
    inc (SpecsChecked);

    {$ifdef Trace}
    Score.ShowPrelim (AvgElementScore/SpecsChecked);
    {$endif}

    ElemSpecs.SetToNext;
End; {While}

AvgElementScore := AvgElementScore/SpecsChecked;
End; {SyntaxSpec.CheckElements}

Procedure SyntaxSpec.CheckSyntax;
var
    EntryPoints: EntryPointList;
    SegMatchSum: word;

    {$ifdef Debug}
    reportstr: string;
    {$endif}

    {$ifdef Trace}
    ElemWin: TextWindow;
    {$endif}
begin
    Failures := 0;
    MatchTotal := 0;

    {$ifdef Trace}
    ElemWin.Init (1,5,18,20,White,Blue,TRUE,'Elements');
    ElemWin.Deactivate;
    {$endif}

    {Loop consists of trying to traverse the syntax tree starting at an }
    {element in the element list. When a failure occurs, try again at the }
    {element after the failed one.}

    Elems.SetToHead;
    Elems.SetToNext;
    While not Elems.AtTail do
    begin
        {$ifdef Trace}
        Elems.Show (ElemWin);
        Score.ShowSyntax (MatchTotal,Failures);
        With Elems.Item^.e do
        begin
            {$ifdef Debug}
            reportstr:= 'About to destroy the EntryPoints List for element: '+
                ClassName[class]+' '+Strung(0,2,durn);
            {$endif}

            Report('Checking syntax starting from '+
                ClassName[class]+' '+Strung(0,2,durn)+' . So far, Fails: '+
                Strung(0,0,Failures)+' '+Match: ' + Strung (0,0,MatchTotal));
        End; {With}
        {$endif}

        {get the entry points for the current element}
        EntryPoints.Init (Elems.Item^.e, ElemSpecs);

        {$ifdef TrackResult}
        writeln (stdout,'| >');
        {$endif}

        {$ifdef Trace}
        EntryWin.Init (19,9,43,9,White,Magenta,TRUE,'Entrypoints');
        EntryPoints.Show (EntryWin);
        {$endif}

        if EntryPoints.Empty
        then begin
            {$ifdef TrackResult}
            with Elems.Item^.e do writeln
            (stdout,'| ',classname[class],' ',durn:4:2,' --- no match');
            {$endif}
        end
    end
end

```

```

        Elems.SetToNext;
        inc(Failures);
    end
else begin
    EntryPoints.SetToEndOfLongestSegment (Elems, SegMatchSum);
    inc(MatchTotal, SegMatchSum);
    if not Elems.AtTail then inc (Failures);
end;

{$ifdef Debug}
Report(reportstr);
{$endif}

EntryPoints.Done (TRUE);

{$ifdef Trace}
EntryWin.Done;
{$endif}
End; {While}

{$ifdef Trace}
ElemWin.Done;
{$endif}
End; {SyntaxSpec.CheckSyntax}

Destructor SyntaxSpec.Done;
begin
    {$ifdef Debug}
    Report ('Destroying a syntax tree');
    {$endif}

    ElemSpecs.SetToHead;
    ElemSpecs.SetToNext;
    While not ElemSpecs.AtTail do with ElemSpecs.Item^ do
    begin
        Nodes.SetToHead;
        Nodes.SetToNext;
        While not Nodes.AtTail do with Nodes.Item^ do
        begin
            {$ifdef Debug}
            Report ('About to destroy the Branches list of syntax tree node '+node^.Identifier);
            {$endif}

            node^.Branches.Done (TRUE);
            dispose (node);
            Nodes.SetToNext;
        end; {While}
        ElemSpecs.SetToNext;
    end; {While}

    {$ifdef Debug}
    Report('About to destroy the Syntax spec''s Element spec list');
    {$endif}

    ElemSpecs.Done (TRUE);
End; {SyntaxSpec.Done}

{=====END OF OBJECTS==}
```

```

VAR
Candidate: SyntaxSpec;      {Why are these global? In case all their space}
Elements: ElementList;     {is allocated on the stack }
```

```

Procedure CheckSyntax;
type
    elementfiletype = file of elementtype;
```

```

var
Candidates: text;
ElementFile: elementfiletype;
CandName: PathStr;
EssentialsPresent: boolean;
FailureScore, MatchScore: word;
PreliminaryScore: real;
OutString: string[80];
tempstr: string[5];
{$ifdef TrackResult}
Dir: DirStr;
```

```

Nam: Dos.NameStr;
Ext: ExtStr;
{$endif}

const
  newline = #13#10;
  dashline = '-----'+
    '-----';

begin
  {$ifdef Trace}
  clrscr;
  {$endif}

  {$ifdef Debug}
  Report ('Memory available at start: '+ Strung(0,0,Memavail));
  {$endif}

  writeln (stdout);

  Elements.Init (ElemFileName); {build an element list from the elements file}

  {$ifdef HumanResult}
  writeln(stdout,'Recognition scores for call ',UpString(ElemFileName));
  writeln(stdout);
  writeln(stdout, 'CANDIDATE',
    '   ESS   PRELIM   SYN FAILS   SYN SCORE');
  writeln(stdout, '=====',
    '   ==   =====   =====   =====');
  {$endif}

  assign(Candidates,CandFileName);
  {$I-} reset(Candidates); {$I+}
  If IOResult <> 0 then
    runerror(UserMessage('Error opening Candidate list file: '+CandFileName));

  {$ifdef TrackResult}
  fsplit (ElemFileName,Dir,Nam,Ext);
  write (stdout, 'Track/s of call in ',UpString(Nam+Ext),
    ' compared with syntax specifications in ');
  fsplit (CandFileName, Dir, Nam, Ext);
  writeln (stdout, Upstring (Nam+Ext),':');
  writeln (stdout);
  {$endif}

  While not eof(Candidates) do      {for each syntax spec,}
  begin
    readln(Candidates,CandName);    {read in the name of the spec (.S) file}
    Candidate.Init(CandName);        {build a syntax tree from that file}

    {$ifdef TrackResult}
    fsplit (CandName, Dir, Nam, Ext);
    Nam := UpString(Nam)+'.S';
    {$endif}

    {$ifdef Trace}
    Score.Init (19,5,20,4,Yellow,Brown,TRUE,Candidate.Birdname);
    {$endif}

    fillchar (OutString[1],80,' ');
    OutString[0] := #80;
    move (Candidate.BirdName[1], OutString[1],length(Candidate.BirdName));

    {$ifdef TrackResult}
    writeln (stdout, 'Syntax spec: ',Nam,' (',Candidate.BirdName,')');
    writeln (stdout, dashline);
    {$endif}

    Candidate.CheckElements (Elements, PreliminaryScore, EssentialsPresent);

    {$ifdef HumanResult}
    str (PreliminaryScore:45:2, tempstr);
    move (tempstr[1],OutString[45],length(tempstr));
    {$endif}

    {$ifdef MachineResult}
    write (stdout, PreliminaryScore:10:2, ' ');
    {$endif}

    {$ifdef Trace}
    Score.ShowPrelim (PreliminaryScore);
    {$endif}
  end

```

```

If EssentialsPresent
then begin
    OutString[40] := 'Y';
    Candidate.CheckSyntax (Elements, FailureScore, MatchScore);

    {$ifdef HumanResult}
    str(FailureScore,tempstr);
    move (tempstr[1], OutString[56],length(tempstr));
    str(MatchScore, tempstr);
    move (tempstr[1], OutString[68],length(tempstr));
    {$endif}

    {$ifdef MachineResult}
    writeln (stdout, FailureScore:10, ' ', MatchScore:10);
    {$endif}

    {$ifdef TrackResult}
    writeln (stdout, '| ');
    writeln (stdout, '| Score for ',Upstring (CandName),'.S:');
    writeln (stdout, '| Failures ',FailureScore,newline,
            '| Match total ',MatchScore);
    {$endif}
end
else begin
    {$ifdef HumanResult}
    OutString[40] := 'N';
    {$endif}

    {$ifdef MachineResult}
    writeln (stdout, 1000:10, ' ',0:10);
    {$endif}

    {$ifdef TrackResult}
    writeln (stdout, '| Essentials not present.');

```

B1.10 MOUSE.PAS

```
{SR-}
UNIT MOUSE;

INTERFACE
USES Dos;
CONST
    lbutton = 0;
    rbutton = 1;
VAR
    mousepresent: boolean;

Procedure ResetMouse;
{Resets the mouse and updates the 'mousepresent' flag}

Procedure ShowMouseCursor;
{Increments the mouse cursor level - zero means visible -
 note that after ResetMouse the cursor level is -1}

Procedure HideMouseCursor;
{Decrements the mouse cursor level - see ShowMouseCursor}

Procedure GetMouseStatus (var x,y: integer; var left,right: boolean);
{Gets current (x,y) pixel position and (left, right) button pressed condition)}

Procedure GetMouseMotion (var x,y: integer);
{Gets (x,y) motion in mickeys (0.01 inch) and resets it to (0,0)}

Procedure GetMousePresses (button: byte; var x, y, presses: integer);
{Gets (x,y) pixel position AT LAST 'button' PRESS, number of presses
 since last call, and current button status. 'button' = 0 means check
 left button, 1 means right. Clears press count.
 Note it is possible to get the current button status from this function,
 by decoding BX - bit 1 is left, bit 2 is right}

Procedure GetMouseReleases (button: byte; var x, y, releases: integer);
{Gets (x,y) pixel position AT LAST 'button' RELEASE, number of presses
 since last call, and current button status. 'button' = 0 means check
 left button, 1 means right. Clears release count.
 Note it is possible to get the current button status from this function,
 by decoding BX - bit 1 is left, bit 2 is right}
```

```
Procedure SetMouseHardCursor (ctype:boolean; set1, set2:word);
{Sets the text cursor to hardware (ctype = true) or software (ctype = false).
 For hardware, set1 is the first cursor line and set2 the last. The cursor is
 on between the first and last lines. CGA - 0..7 lines. MDA - 0..11 lines.
 For software, set1 is the screen mask (gets ANDed with the screen attribute)
 and set2 is the cursor mask (gets XORed with the attribute after the AND).}
```

For reference, here is the 16 bit attribute for text mode:

MSB																LSB
b1	bk	bk	bk	in	fr	fr	fr	ch	ch	ch	ch	ch	ch	ch	ch	ch

where b1 = blinking, bk = background colour, in = intensity, fr = foreground colour and ch = ASCII code}

```
type proctype = procedure;
Procedure SetMouseInterrupt (CondMask:word; Proc:proctype);
{Sets up an interrupt procedure ('Proc' which must be far)
 which is called whenever the conditions dictated by the set bits of
 'CondMask' occur:
      CondMask: 7  6  5  4  3  2  1  0  -- CURSOR POSTION CHANGED
                X  X  X  |  |  |  |
                |  |  |  |  |  |  |
                |  |  |  |  |  |  | L BUTTON PRESSED
                |  |  |  |  |  |  |
                |  |  |  |  |  |  | L BUTTON RELEASED
                |  |  |  |  |  |  |
                |  |  |  |  |  |  | R.BUTTON PRESSED
                |  |  |  |  |  |  |
                R BUTTON RELEASED
```

NOTES: Usage is SetMouseInterrupt (mask,MyProc) where MyProc is an INTERFACE procedure or is compiled with \$F+. It is also likely that stack-checking will have to be turned off (\$S-) as explained in RestoreTurboDseg.

Always call SetMouseInterrupt(0,MyProc) before exiting the program or the address will continue to be called even if the proc is no longer in memory

The called procedure (eg 'MyProc' above) can access some mouse parameters at the time of the call without using a mouse funtion call by getting the register values (eg. using the GetRegs procedure described later).

The parameters are:

AX: condition mask - whichever condition occurred sets a bit according to the definition of 'CondMask' above.

BX: button state

CX: x cursor position

DX: y cursor position

See RestoreTurboDseg for an example of a mouse interrupt routine.

}

Procedure GetRegs (var regs: registers);

{Returns value of all the registers at the time of calling - there is no guarantee that they will still be the same after the procedure returns returns.}

Procedure PushDseg; inline (\$1E); {push ds}

Procedure PopDseg; inline (\$1F); {pop ds}

{Used to save the interrupt routine's data segment, which is different from Turbo's - see RestoreTurboDseg below}

Procedure RestoreTurboDseg;

{Must be called after saving the Data Segment register (PushDseg) in a mouse interrupt routine. This procedure wont work unless SetUpDsegRestore has been called from outside the interrupt routine - but that is taken care of by this unit's initialization.

Here is a shell mouse interrupt routine:

(\$F+) (\$S-)

Procedure MSInt;

Begin

PushDseg;

RestoreTurboDseg;

...

any code

...

...

PopDseg;

End;

(\$F+) (\$S-)

Failure to save and restore the DS register as above will result in unpredictable behaviour when global variables are accessed. (Thanks, Microsoft). Any code which is called before RestoreTurboDseg (such as Turbo Pascal Stack Checking code) will execute incorrectly if it references things via DS. In the case of stack checking, this can cause an unjustified termination due to "stack overflow" because the maximum stack size is stored in a variable which is offset from the Turbo DS. In this case the solution is simply to turn off stack-checking for that procedure with (\$S-).

}

Function MouseAction: boolean;

{returns true if the mouse has moved since the last call to MouseAction or GetMouseMotion or if any of the buttons are currently being pressed}

IMPLEMENTATION

(\$S-)

VAR MouseDS: word; {Used by SaveMouseDseg and RestoreMouseDseg to save the mouse driver's data segment}

Procedure ResetMouse;

var regs: registers;

begin

regs.ax := 0; {Mouse Installed Flag and Reset function}

Intr (\$33, regs);

mousepresent := regs.ax <> 0;

End;

Procedure ShowMouseCursor;

var regs: registers;

begin

regs.ax := 1; {Show Cursor function};

Intr (\$33, regs);

End;

```

Procedure HideMouseCursor;
var regs: registers;
begin
    regs.ax := 2; {Hide Cursor function};
    Intr ($33, regs);
End;

Procedure GetMouseStatus (var x, y: integer; var left, right: boolean);
var regs: registers;
begin
    regs.ax := 3; {Get Mouse Position and Button Status}
    Intr ($33, regs);
    left := (regs.bx and $0001) <> 0;
    right := (regs.bx and $0002) <> 0;
    x := integer(regs.cx);
    y := integer(regs.dx);
End;

Procedure GetMousePresses (button: byte;
                           var x,y, presses: integer);
var regs: registers;
begin
    regs.ax := 5; {Get Button Press Information}
    regs.bx := button;
    Intr ($33, regs);
    presses := integer(regs.bx);
    x := integer(regs.cx);
    y := integer(regs.dx);
End;

Procedure GetMouseReleases (button: byte;
                            var x,y, releases: integer);
var regs: registers;
begin
    regs.ax := 6; {Get Button release Information}
    regs.bx := button;
    Intr ($33, regs);
    releases := integer(regs.bx);
    x := integer(regs.cx);
    y := integer(regs.dx);
End;

Procedure GetMouseMotion (var x,y: integer);
var regs: registers;
begin
    regs.ax := 11; {Read Mouse Motion Counters function}
    Intr($33, regs);
    x := integer(regs.cx);
    y := integer(regs.dx);
End;

Procedure SetMouseHardCursor;
var regs: registers;
begin
    regs.ax := 10; {Set Text Cursor function}
    regs.bx := byte(ctype);
    regs.cx := set1;
    regs.dx := set2;

    Intr($33, regs);
End;

Procedure SetMouseInterrupt (CondMask:word; Proc:proctype);
var regs: registers;
begin
    regs.ax := 12; {Set User-Defined Subroutin Input Mask function}
    regs.cx := CondMask;
    regs.dx := OfS(Proc);
    regs.es := Seg(Proc); {es:dx points to the interrupt procedure}

    Intr($33, regs);
end;

Procedure GetRegs (var regs:registers);
external; {$L GETREGS.OBJ} {FAR PROC}

Procedure RestoreTurboDseg;
begin
    inline ($B8/$01/$23/ {mov ax,2301h} {The value 2301 is a placeholder which
                                     is replaced AT RUN TIME with the Data
                                     segment value by the SetUpDsegRestore
                                     procedure})

```

```

        $90/          {nop}
        $8E/$D8);    {mov ds,ax}
End(Procedure);

Procedure SetUpDsegRestore;
type
    wordptr = ^word;
    longptr = ^longint;
var  procofs,procseg: word;

begin
    procseg := seg(restoreturbodseg);
    procofs := ofs(restoreturbodseg);

    while not ((Mem[procseg:procofs] = $B8) and (Mem[procseg:procofs+3] = $90))
    do inc(procofs);
    inc(procofs);

    MemW[procseg:procofs] := Dseg;
End;

Function MouseAction: boolean;
var
    x,y: integer;
    mleft, mright: boolean;
begin
    GetMouseStatus (x,y, mleft, mright);
    GetMouseMotion (x,y);
    MouseAction := (x + y <> 0) or mleft or mright ;
End;

{$S+}

BEGIN
    SetUpDsegRestore;
END(UNIT).

```

B1.10.1 GETREGS.ASM

```
.MODEL TPASCAL
.CODE
IDEAL

;NB .MODEL TPASCAL AUTOMATICALLY MAKES THE SPACE FOR LOCAL VARIABLES, IE.
;IT ASSEMBLES sub sp,localspace WHICH IS NOT CLEAR FROM THE DOCUMENTATION
;SOME OTHER TPASCAL MAGIC:
; AT BEGINNING - push bp, mov bp,sp
; AT END - mov sp,bp, pop bp, ret <value that cleans up parameters>
;SO STACK IS AUTOMATICALLY CLEANED UP - AS LONG AS YOU DONT CHANGE BP
;AND DONT POP THE STORED bp OR THE PARAMETERS OFF - INDEX THEM. ANY
;LOCAL USE YOU MAKE OF THE STACK IS ALSO CLEANED UP.
;ALL THATS LEFT FOR YOU TO DO IS MAKE SURE ds IS @data ON RETURN

;Procedure GetRegs (Regs: Registers);
;In MOUSE.PAS

;Procedure to return the current values of all the registers (except Flags).

PROC GetRegs FAR Regs:dword ;Regs points to a Turbo Pascal Registers record
LOCAL SAVEAX:word, SAVEBX:word
PUBLIC GetRegs

    mov [SAVEAX],ax          ;save ax value
    mov [SAVEBX],bx          ;save bx value
    mov ax, [WORD Regs+2]
    mov ds, ax                ;ds holds the segment part of Regs
    ASSUME cs:CODE,ds:nothing
    mov bx, [WORD Regs]        ;ds:bx points to start of Regs

    ;The above access is via BP segment register. The following is via DS

    mov ax,[SAVEAX]
    mov [bx],ax
    mov ax, [SAVEBX]
    mov [bx+2],ax
    mov [bx+4],cx
    mov [bx+6],dx
    mov [bx+8],bp
    mov [bx+10],si
    mov [bx+12],di
    mov [WORD bx+14],@data      ;the DS value before it was changed
    mov [bx+16],es
    mov [WORD bx+18],0          ;what must I do about flags.

    mov ax,@Data
    mov ds,ax
    ret
ENDP GetRegs

END
```

B1.11 ICON.PAS

```
UNIT icon;

INTERFACE

TYPE charset = set of char;

Procedure GetGraphInput (var px,py:integer; var Inkey:char; InKeySet:charset;
                        quickstep: byte; waitforkey: boolean;
                        leftlimit,toplimit,rightlimit,bottomlimit:integer);
{
  [place an icon at (x,y) and allow user to move it around on the screen
   within the rectangle defined by the XXXlimit parameters]
  [the cursor keys move in steps of one pixel. The number keys (shifted
   keypad cursor keys) move in steps of 'quickstep']
  [If 'waitforkey' is true, return only when a key in 'InKeySet' is pressed.
   If 'waitforkey' is false, return also when there is any motion of the icon.
   On return 'x' and 'y' point to the last icon position and 'InKey' is
   equal to the key pressed, #0 if the exit was not due to a keypress, #1 or
   #2 for LEFT or RIGHT mouse clicks respectively.]
}

Procedure ChooseBox (var xlbox,ylbox,x2box,y2box:integer;
                    var status:boolean);
{
  [allow interactive choice of screen area, returning top left corner
   in (xlbox,ylbox) and bottom right corner in (x2box,y2box). Start off
   with a box defined by the passed-in values of xl,yl,x2 and y2 and let
   "Tab" key mean "switch activity to opposite corner", "Enter" mean "return
   new box coords, with 'status' = TRUE and "Esc" mean "return old box coords
   with 'status' = FALSE. Mouse input works with double clicks:
   LEFT,LEFT = "Enter", RIGHT,RIGHT = "Esc" and LEFT,RIGHT or
   RIGHT,LEFT = "Tab". In all cases the mouse should be motionless between
   clicks, otherwise it is as if no key was pressed]
}

VAR
  MenuQuitFlag: boolean;    {Used by caller to break out of menu}

CONST WidthMenu = 8*16;      {Width in pixels of the menu}
      MaxNumItems = 10;

TYPE
  ItemName = string[WidthMenu div 8];
  NamePtr = ^ItemName;
  NoParamProc = Procedure;
  numitemtype = 1..MaxNumItems;

Procedure null; {for unassigned procedures in a call to Menu}

Procedure Menu (xmenu, ymenu: word; numitems: numitemtype;
               name0:NamePtr; proc0:NoParamProc;
               name1:NamePtr; proc1:NoParamProc;
               name2:NamePtr; proc2:NoParamProc;
               name3:NamePtr; proc3:NoParamProc;
               name4:NamePtr; proc4:NoParamProc;
               name5:NamePtr; proc5:NoParamProc;
               name6:NamePtr; proc6:NoParamProc;
               name7:NamePtr; proc7:NoParamProc;
               name8:NamePtr; proc8:NoParamProc;
               name9:NamePtr; proc9:NoParamProc);
{
  REPEAT
    [save the part of the picture over which the menu is to be drawn]

    [place a menu with top left corner at (xmenu,ymenu) containing the names
     in the 'nameX' parameters. Only the first 'numitems' names are shown.]

    [allow choice of a menu option using the arrow icon - "Enter" selects
     the item currently pointed to, ESC exits the menu, A to Z choose and
     immediate execute the first item beginning with that letter.]

    [erase the menu, restoring the picture underneath]

    IF [ESC was not pressed]
    THEN [call the procedure "procX" associated with the chosen "nameX"]
    ENDIF
  UNTIL [ESC is pressed]

  With the mouse, LEFT = ENTER and RIGHT = ESC.
}
}
```

```

CONST MaxLineLength = 78;
TYPE linestring = string[MaxLineLength];

Function Prompt (xprompt, yprompt: word; Message, Default:linestring;
                NumChars:byte; var Escaped: boolean):linestring;
{
  [calculate the size of the prompter and adjust its position if it
   wont fit on the screen]
  [save the picture under the place where the prompt will be]

  [read in and return a string of maximum length specified by 'NumChars'.
   The case of 'NumChars' = 1 is handled seperately - no terminating 'Enter'
   is required, 'escaped' and 'default' are not used]
}
{
  'Default' is the default string that will be placed in the input edit
  buffer in the prompt window. If escape is pressed at any stage, input
  terminates, the default value is returned and 'Escaped' becomes true.
}

```

IMPLEMENTATION

USES Dos, Crt, Graph, MOUSE;

TYPE

```

  IconType = record
    wid, len: word;
    pic: array [1..64] of byte;
  end;

```

VAR

```

  Driver, Mode: integer;
  i, j: longint;
  SpecialKey: boolean;
  x, y, oldx, oldy: integer;
  UnderIcon, PosPointIcon, NegPointIcon, Temp: IconType;
  pal: palettetype;

```

Procedure GetGraphInput;

const left = 0; right = 1;

var ch: char;

```

  viewport: viewporttype;
  mx, my, presses: integer;
  startx, starty: integer;

```

begin

```

  oldx := px; oldy := py; startx := px; starty := py;
  {place pointer in first position}
  getimage (px, py, px+PosPointIcon.wid, py+PosPointIcon.len, UnderIcon);
  putimage (px, py, NegPointIcon, AndPut);    {put the pointer icon}
  putimage (px, py, PosPointIcon, OrPut);     {at (px,py)}

```

```

  GetMouseMotion (mx, my);                    {clear the mouse counters}
  GetMousePresses (left, mx, my, presses);
  GetMousePresses (right, mx, my, presses);
  ch := #0;

```

repeat

 If mousepresent then

 begin

```

      GetMouseMotion (mx, my);
      inc (px, mx);
      inc (py, my);

```

```

      GetMousePresses (right, mx, my, presses);

```

```

      if presses <> 0 then ch := #2;

```

```

      GetMousePresses (left, mx, my, presses);

```

```

      if presses <> 0 then ch := #1;

```

 End{If};

 If keypressed then

 begin

```

      ch := readkey;

```

```

      if ch = #0 then begin SpecialKey := TRUE; ch := readkey end
                     else SpecialKey := FALSE;

```

 if SpecialKey

 then Case ch of

```

      #77: inc(px);           {RIGHT_ARROW}
      #75: dec(px);           {LEFT_ARROW}
      #72: dec(py);           {UP_ARROW}
      #80: inc(py);           {DOWN_ARROW}

```

 End{Case}

 else Case ch of

```

      '6': inc(px, quickstep); {SHFT RIGHT_ARROW}
      '4': dec(px, quickstep); {SHFT LEFT_ARROW}
      '8': dec (py, quickstep); {SHFT UP_ARROW}
      '2': inc (py, quickstep); {SHFT DOWN_ARROW}

```

```

        End{Case};
        if ch in [#77,#75,#72,#80,'6','4','8','2'] then ch := #0;
        {pretend no particular key has been pressed}
    End{If};

    If (px <> oldx) or (py <> oldy) then
    begin
        {check requested limits}
        if px < leftlimit then px := leftlimit;
        if px > rightlimit then px := rightlimit;
        if py < toplimit then py := toplimit;
        if py > bottomlimit then py := bottomlimit;

        {check for physical limits of viewport left corner-
        in case the requested limits are wrong}
        getviewsettings (viewport);
        with viewport do if px > x2-x1-7 then px := x2-x1-7;
        if px < 0 then px := 0;
        with viewport do if py > y2-y1 then py := y2-y1;
        if py < 0 then py := 0;

        {replace the picture under the pointer,  }
        {and save the part under the next position}
        putimage (oldx,oldy, UnderIcon, normalput);
        getimage (px, py, px+PosPointIcon.wid, py+PosPointIcon.len, UnderIcon);
        oldx := px; oldy := py;

        PutImage (px, py, NegPointIcon, AndPut);      {put the pointer icon}
        PutImage (px, py, PosPointIcon, OrPut);        {at (px,py)}
    End{If};

    until (ch in InKeyset) or
        ((not waitforkey) and ((px-startx <> 0) or (py-starty <> 0)));

    InKey := ch;

    PutImage (px, py, UnderIcon, CopyPut);             {remove pointer}
End{Procedure};

Procedure ChooseBox;
type  imagearray = array[1..1400] of byte; {Enough for a single column
                                           (8 pixels) of a EGA hires screen}
const
    TopLeft = FALSE;
    BotRight = TRUE;

var x,y, dummyx, dummyy, presses: integer;
    corners: array [TopLeft..BotRight] of record x,y:integer end;
    ch, ch2:char;
    corner, mleft, mright:boolean;
    LeftSave,RightSave,TopSave,BottomSave:^imagearray;

begin
    new (TopSave);
    new (BottomSave);
    new (LeftSave);
    new (RightSave);

    x := 0; y := 0;
    corners[TopLeft].x := xlbox;
    corners[TopLeft].y := ylbox;
    corners[BotRight].x := x2box;
    corners[BotRight].y := y2box;
    corner := BotRight;
    ch := #13;

    getimage(corners[TopLeft].x,corners[TopLeft].y,
        corners[BotRight].x,corners[TopLeft].y+1, TopSave^);
    getimage(corners[TopLeft].x,corners[BotRight].y-1,
        corners [BotRight].x, corners[BotRight].y,BottomSave^);
    getimage(corners[TopLeft].x, corners[TopLeft].y,
        corners[TopLeft].x+1, corners[BotRight].y, LeftSave^);
    getimage(corners[BotRight].x-1, corners[TopLeft].y,
        corners[BotRight].x, corners[BotRight].y, RightSave^);
    {Save new boxed area}

```

```

setcolor(white);
rectangle (corners[TopLeft].x,corners[TopLeft].y,
           corners[BotRight].x,corners[BotRight].y);
setcolor(black);
rectangle (corners[TopLeft].x+1,corners[TopLeft].y+1,
           corners[BotRight].x-1,corners[BotRight].y-1);
{Draw new box}

Repeat
  x := corners[corner].x;
  y := corners[corner].y;
  ch := #0;

  If mousepresent then
  begin
    GetMousePresses (lbutton, dummyx, dummyy,presses);
    If presses <> 0 then ch := #1;
    GetMousePresses (rbutton, dummyx, dummyy,presses);
    If presses <> 0 then ch := #2;
  End{If};

  If ch = #0 then
  Case corner of
    BotRight: GetGraphInput
      (x,y,ch,[#1,#2,#9,#13,#27],
       10,false,
       corners[TopLeft].x+2,corners[TopLeft].y+2,getmaxx,getmaxy);
    TopLeft: GetGraphInput
      (x,y,ch,[#1,#2,#9,#13,#27],
       10,false,
       0,0,corners[BotRight].x-2,corners[BotRight].y-2);
  End{Case};

  {If ch = #1 then ch := #13;}

  If not (ch in [#1,#2,#9,#13,#27]) then
  begin
    putimage(corners[TopLeft].x,corners[TopLeft].y,TopSave^,NormalPut);
    putimage(corners[TopLeft].x,corners[BotRight].y-1,BottomSave^,NormalPut);
    putimage(corners[TopLeft].x, corners[TopLeft].y, LeftSave^,NormalPut);
    putimage(corners[BotRight].x-1, corners[TopLeft].y, RightSave^,NormalPut);

    corners[corner].x := x;
    corners[corner].y := y;

    getimage(corners[TopLeft].x,corners[TopLeft].y,
             corners[BotRight].x,corners[TopLeft].y+1, TopSave^);
    getimage(corners[TopLeft].x,corners[BotRight].y-1,
             corners [BotRight].x, corners[BotRight].y,BottomSave^);
    getimage(corners[TopLeft].x, corners[TopLeft].y,
             corners[TopLeft].x+1, corners[BotRight].y, LeftSave^);
    getimage(corners[BotRight].x-1, corners[TopLeft].y,
             corners[BotRight].x, corners[BotRight].y, RightSave^);
    {Save new boxed area}

    setcolor(white);
    rectangle (corners[TopLeft].x,corners[TopLeft].y,
              corners[BotRight].x,corners[BotRight].y);
    setcolor(black);
    rectangle (corners[TopLeft].x+1,corners[TopLeft].y+1,
              corners[BotRight].x-1,corners[BotRight].y-1);
    {Draw new box}
  End{If};

  {
  The following block means: (L,R = left and right mouse buttons)
  LL = ENTER (accept box)
  RR = ESC (cancel choosing)
  LR = RL = TAB (swap corners)
  L<key> = R<key> = <key> ( ignore single ... )
  L<movement> = R<movement> = <movement> ( ... mouse presses )
  }

  If ch in [#1,#2] then
  begin
    GetGraphInput(x,y,ch2,[#1,#2,#9,#13,#27],10,false,x-1,y-1,x+1,y+1);
    if ch2 = #1
    then if ch = #1
         then ch := #13
         else ch := #9
    else if ch2 = #2
         then if ch = #2
              then ch := #27
              else ch := #9
         else ch := ch2;
  End{If};

```

```

    if ch = #9 then corner := not corner;

Until ch in [#13,#27];

putimage(corners[TopLeft].x, corners[TopLeft].y, TopSave^, NormalPut);
putimage(corners[TopLeft].x, corners[BotRight].y-1, BottomSave^, NormalPut);
putimage(corners[TopLeft].x, corners[TopLeft].y, LeftSave^, NormalPut);
putimage(corners[BotRight].x-1, corners[TopLeft].y, RightSave^, NormalPut);
{Erase previous box}

dispose (RightSave);
dispose (LeftSave);
dispose (BottomSave);
dispose (TopSave);

if ch = #13
Then begin
    x1box := corners[TopLeft].x;
    y1box := corners[TopLeft].y;
    x2box := corners[BotRight].x;
    y2box := corners[BotRight].y;
    status := TRUE;
End(Then)
else status := FALSE;

if mousepresent then repeat
    GetMouseStatus (dummyx, dummyy, mleft, mright)
    until not (mleft or mright);

End(Procedure);

Procedure null;
begin
end;

Procedure Menu;
const
    lengthitem = 11;

type
    menuimage = array [1..WidthMenu*200 div 2] of byte;
    {image storage space for an EGA area of up to WidthMenu x 200}
var
    UnderMenu: ^menuimage;
    Choice: 0..MaxNumItems-1;
    lengthmenu: 0..200;
    itemcount: 1..MaxNumItems;
    np: NoParamProc;
    ItemTable: array [0..MaxNumItems-1] of record
        name: NamePtr;
        proc: NoParamProc;
    end;

    x, y: integer;
    ch: char;
    selected: boolean;

begin
    new (UnderMenu);

    ItemTable[0].name := name0; ItemTable[0].proc := proc0;
    ItemTable[1].name := name1; ItemTable[1].proc := proc1;
    ItemTable[2].name := name2; ItemTable[2].proc := proc2;
    ItemTable[3].name := name3; ItemTable[3].proc := proc3;
    ItemTable[4].name := name4; ItemTable[4].proc := proc4;
    ItemTable[5].name := name5; ItemTable[5].proc := proc5;
    ItemTable[6].name := name6; ItemTable[6].proc := proc6;
    ItemTable[7].name := name7; ItemTable[7].proc := proc7;
    ItemTable[8].name := name8; ItemTable[8].proc := proc8;
    ItemTable[9].name := name9; ItemTable[9].proc := proc9;

    lengthmenu := numitems*lengthitem; {length is down the screen}
    if ymenu+lengthmenu > getmaxy then ymenu := getmaxy-lengthmenu;
    if xmenu+widthmenu > getmaxx then xmenu := getmaxx-widthmenu;

    ICON.MenuQuitFlag := FALSE;
    Choice := 0;
    x := widthmenu - 9;
    y := Choice*lengthitem+3;
    Repeat
        {save whatever is under the menu}
        getimage (xmenu, ymenu, xmenu+widthmenu, ymenu+lengthmenu, UnderMenu^);

```

```

    {draw the menu}
    setcolor (white);
    settextjustify (lefttext,bottomtext);
    setviewport (xmenu,ymenu,xmenu+widthmenu,ymenu+lengthmenu, clipon);
    clearviewport;
    rectangle (0,0,widthmenu,lengthmenu);
    For itemcount := 1 to numitems do
    begin
        moveto (2,itemcount*lengthitem);
        outtext(ItemTable[itemcount-1].name^);
        moveto (0,itemcount*lengthitem);
        linerel (widthmenu,0);
    End{For};
    settextjustify (LeftText,TopText);

    {get the choice}
    GetGraphInput (x,y,ch, [#1,#2,#13,#27,'a'..'z','A'..'Z'],
        lengthitem,true,
        widthmenu-24,3,widthmenu-9,lengthmenu-8);
    Choice := y div lengthitem; {Choose according to position}
    If upcase(ch) in ['A'..'Z'] then
    begin
        itemcount := 1;
        while (upcase(ch) <> upcase (ItemTable[itemcount-1].Name^[1]))
            and (itemcount <= numitems)
        do inc (itemcount);
        if itemcount <= numitems
        then begin
            Choice := itemcount-1;
            y := (itemcount-1)*lengthitem+2;
            ch := #13; {pretend pointer was moved and enter pressed}
        end;
    End{If};
    if ch = #1 then ch := #13;
    if ch = #2 then ch := #27;

    putimage (0,0,UnderMenu^,NormalPut);
    {remove the menu and restore the picture}
    setviewport (0,0,getmaxx,getmaxy,clipon);

    if ch = #13 then ItemTable[Choice].proc;
    {if an action was selected rather than ESC or a letter,
     execute the chosen action}
    Until (ch = #27) or ICON.MenuQuitFlag;

    MenuQuitFlag := FALSE;

    dispose (UnderMenu);
End{Procedure};

Function Prompt;

const boxdepthc = 20;
var boxwidth: 0..639;
    boxdepth: 0..boxdepthc;
    textinfo: textsettingstype;

type promptimage = array [1..MaxLineLength*8*boxdepthc div 2] of byte;
                                {stores two lines in EGA graphics modes}
var UnderPrompt: ^promptimage;
    instrin: linestring;
    ch: char;
    mx, my: integer;
    mleft, mright,gotone: boolean;
    count:word;
    oldviewport: viewporttype;

{}Function ReadInString(numchars:byte):linestring;
{}
{} Read numchars characters, with flashing cursor and backspacing,
{} in graphics mode. The echoed input starts at the current
{} graphics cursor position. Pressing 'ESC' returns the default string.
{} with 'Escaped' true.
{}
{}const cursor: array[1..36] of byte = (7,0,7,0,
{}                                     $FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF,
{}                                     $FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF,
{}                                     $FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF,
{}                                     $FF,$FF,$FF,$FF,$FF,$FF,$FF,$FF);
{}
{}    blank: array[1..36] of byte = (7,0,7,0,
{}                                   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
{}                                   0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0);
{}var firstkey: boolean;

```

```

{} strindex:=byte;
{} savex, savey:= integer;
{}
{}begin
{} firstkey := TRUE;
{} instring := Default;
{} strindex := length (Default);
{} {putimage (getx,gety,Blank,XorPut);}
{} Repeat {accept the string}
{} gotone := FALSE;
{} Repeat {flash cursor}
{} putimage (getx,gety,Cursor,XorPut);
{} count := 0;
{} While (not gotone) and (count < 1000) do
{} begin
{} GetMouseStatus (mx, my, mleft, mright);
{} if (mousepresent and (mleft or mright)) or keypressed then
{} begin
{} if keypressed then ch := readkey;
{} if mousepresent and mleft then ch := #13;
{} if mousepresent and mright then ch := #27;
{} gotone := TRUE;
{} end;
{}
{} inc(count);
{} End{While};
{} putimage (getx,gety,Cursor,XorPut);
{} count := 0;
{} While (not gotone) and (count < 1000) do
{} begin
{} GetMouseStatus (mx, my, mleft, mright);
{} if (mousepresent and (mleft or mright)) or keypressed then
{} begin
{} if keypressed then ch := readkey;
{} if mousepresent and mleft then ch := #13;
{} if mousepresent and mright then ch := #27;
{} gotone := TRUE;
{} end;
{} inc(count);
{} End{While};
{} Until gotone;
{} {putimage (getx,gety,Blank,XorPut);}
{}
{} if (not (ch in [#13,#8,#0]))
{} Then begin
{} if firstkey then
{} for i := 1 to length(instring) do
{} begin
{} moverel (-8,0);
{} putimage (getx,gety,Blank,NormalPut);
{} delete (instring,length(instring),1);
{} strindex := 0;
{} end;
{} if (length(instring) < numchars) then
{} begin
{} inc(strindex);
{} insert(ch,instring,strindex);
{} savex := getx; savey := gety;
{} for i := strindex to length (instring) do
{} begin
{} putimage (getx,gety,Blank,NormalPut);
{} outtext (instring[i]);
{} end;
{} moveto (savex+8, savey);
{} end;
{} End{Then}
{}
{} Else begin
{} if ch = #0 then ch := readkey;
{} Case ch of
{} #8: If strindex > 0 then
{} begin
{} moverel (-8,0);
{} delete (instring,strindex,1);
{} savex := getx; savey := gety;
{} for i := strindex to length (instring) do
{} begin
{} putimage (getx,gety,Blank,NormalPut);
{} outtext (instring[i]);
{} end;
{} putimage (getx,gety,Blank,NormalPut);
{} moveto(savex, savey);
{} dec (strindex);
{} End{If};
{} #75:If strindex > 0 then

```

```

()      begin
()      moverel (-8,0);
()      dec(strindex);
()      End{If};
()      #77:If strindex < length(instring) then
()      begin
()      moverel (8,0);
()      inc (strindex);
()      End{If};
()      #83:If strindex < length(instring) then
()      begin
()      delete (instring,strindex+1,1);
()      savex := getx; savey := gety;
()      for i := strindex+1 to length (instring) do
()      begin
()      putimage (getx,gety,Blank,NormalPut);
()      outtext (instring[i]);
()      end;
()      putimage (getx,gety,Blank,NormalPut);
()      moveto(savex, savey);
()      End{If};
()      End{Case};
()      End{Else};
()      firstkey := FALSE;
()      Until ch in [#13,#27];
()
()      Escaped:= FALSE;
()      If ch = #13 then ReadInString := instring
()      else begin
()      ReadInString := Default;
()      Escaped := TRUE;
()      end;
()      End{Function};

begin
  new (UnderPrompt);

  gettextsettings (textinfo);
  setttextstyle (defaultfont, horizdir, 1);
  setttextjustify (lefttext, toptext);

  boxwidth := (length(Message)+2)*8-1;
  if Numchars = 1 then boxdepth := boxdepthc div 2  {no echo in this case}
  else boxdepth := boxdepthc;

  if NumChars >= MaxLineLength-1 then NumChars := MaxLineLength-1;
  if boxwidth < (NumChars+2)*8-1 then boxwidth := (NumChars+2)*8-1;
  {need at least 2 characters extra to prevent erasing the end of the box}

  if boxwidth > getmaxx then boxwidth := getmaxx;
  if xprompt > getmaxx-boxwidth then xprompt := getmaxx-boxwidth;
  if yprompt > getmaxy-boxdepth then yprompt := getmaxy-boxdepth;

  getviewsettings(oldviewport);

  setcolor (white);
  setviewport (xprompt,yprompt,xprompt+boxwidth,yprompt+boxdepth, clipon);
  getimage (0,0,boxwidth,boxdepth, UnderPrompt^);
  clearviewport;
  rectangle (0,0,boxwidth,boxdepth);
  outtextxy (4,2,Message);
  moveto (4,10);
  outtext (Default);

  if NumChars = 1 then begin
    repeat
      GetMouseStatus (mx, my, mleft, mright);
    until not (mleft or mright) or not mousepresent;
    repeat
      GetMouseStatus (mx, my, mleft, mright);
    until keypressed or (mousepresent and (mleft or mright));
    if keypressed then instring := readkey;
    if mousepresent and mleft then instring := #1;
    if mousepresent and mright then instring := #2;
  end
  else instring := ReadInString(NumChars);

  putimage (0,0,UnderPrompt^,NormalPut);
  dispose (UnderPrompt);
  with oldviewport do setviewport (x1,y1,x2,y2,clip);
  Prompt := instring;

```

```

With textinfo do
begin
  setttextjustify (horiz, vert);
  setttextstyle (font, direction, charsize);
End(With);
End(Function);

```

```

BEGIN {ICON SHAPE DEFINITION}
  driver := detect;
  initgraph (driver, mode, 'c:\t\bgi');

  ResetMouse;

  x := getmaxx div 2; y := getmaxy div 2;
  oldx := x; oldy := y;
  getpalette(pal);

  (define icon size)
  PosPointIcon.wid := 7;
  PosPointIcon.len := 8;
  NegPointIcon.wid := PosPointIcon.wid;
  NegPointIcon.len := PosPointIcon.len;
  UnderIcon.wid := PosPointIcon.wid;
  UnderIcon.len := PosPointIcon.len;
  Temp.wid := PosPointIcon.wid;
  Temp.len := PosPointIcon.len;
  getimage (0, 0, PosPointIcon.wid, PosPointIcon.Len, UnderIcon); {save work space}
  for i := 1 to 64 do Temp.pic[i] := 0;
  putimage (0, 0, Temp, NormalPut); {clear work space}
  getpalette (pal); {save palette}
  setpalette (15,63); {use white to draw}
  setcolor(15);

  moveto (1,0); {draw positive part}
  lineto (7,3); {of pointer icon}
  moveto (5,4);
  lineto (7,6);
  lineto (7,7);
  lineto (5,7);
  moveto (4,7);
  lineto (2,5);
  moveto (1,6);
  lineto (0,6);
  lineto (0,1);

  {store positive pointer icon}
  getimage (0, 0, PosPointIcon.wid, PosPointIcon.Len, PosPointIcon);

  putimage (0, 0, Temp, NormalPut); {clear work space}
  moveto (0,0); {draw negative part}
  lineto (6,6); {of pointer icon}
  moveto (6,7);
  lineto (1,2);
  lineto (1,5);
  moveto (2,4);
  lineto (4,6);
  moveto (3,2);
  lineto (4,3);
  lineto (5,3);

  {store negative pointer icon}
  getimage (0, 0, PosPointIcon.wid, PosPointIcon.Len, NegPointIcon);
  for i := 1 to 64 do NegPointIcon.pic[i] := NOT (NegPointIcon.pic[i]);

  setpalette (15, pal.colors[15]); {restore palette}
  putimage (0, 0, UnderIcon, NormalPut); {restore workspace}
  CloseGraph;

END(UNIT).

```

B1.12 PRTEGA.PAS

```
UNIT PrtEga;
{
  Provides a program with the ability to perform a screen dump of EGA
  and VGA screens (16 colour modes). Colours are handled in a more
  sophisticated way than most dumps, and most common printer types are
  catered for.
}

INTERFACE

type
  GreyPaletteType = array [0..15] of 0..15;
  PrinterType = (Epson, LaserJet2, Postscript);
  PrintFormType = (Full, Split);
  PrintScaleType = (Normal, Squashed, Stretched);
  GreyPaletteChoice = (AsQuantized, Differentiated);

var  GreyPalette: GreyPaletteType;    {holds colour-to-grey map}

Procedure InvertGreyPalette;
Procedure ChooseGreyPalette (choice: GreyPaletteChoice);
Procedure PrintEga (DumpDestStr:string;
                   PrinterPar:PrinterType;
                   PrintScalePar:PrintScaleType;
                   PrintFormPar:PrintFormType;
                   x1,y1, x2,y2: integer);

IMPLEMENTATION
uses Graph (* ,ICON *);
const
  MAXROWPIX = 640;
  MAXCOLPIX = 480;

type
  BitCounter = 0..7;
  ColourIndex = 0..15;
  ColourByte = array [BitCounter] of ColourIndex;
  PtrColourByte = ^ColourByte;

  ColSize = 1..MAXCOLPIX;
  EgaByte = array[1..4] of byte;
  EgaCol = array [ColSize] of EgaByte;
  EgaImageCol = record
    width,length: word;
    pixels: EgaCol;
  end;

  HeightType = (Above,Below);
  SideType = (Left,MidLeft, MidRight,Right);
  LayerType = (Layer1,Layer2);
  Epson8Dot = array [Layer1..Layer2, Left..Right, Above..Below] of byte;
  Epson8DotLine = array [ColSize] of Epson8Dot;

  LJ2DotPair = array [1..4] of byte;
  LJ2DotGroup = array [BitCounter] of LJ2DotPair;
  LJ2DotLine = array [1..MAXCOLPIX] of LJ2DotGroup;

var  DumpDest: text;                {sets dump destination (printer/disk)}
     Printer: PrinterType;
     PrintScale:PrintScaleType;
     PrintForm: PrintFormType;

procedure GetColourByte (dots:EgaByte;                {input image bytes}
                        var colours: ColourByte);      {output image byte colours}
{
  Obtain the colour index (0 to 15)
  of the pixels in a byte on the
  screen (actually four bytes --
  "overlayed" on top of each other.
  In planes. The first plane gives
  the LSB of the colour index number).
}
{
var
  bit: BitCounter;
  plane: 1..4;
  colour: 0..15;
```

```

begin
  for bit := 0 to 7 do
    begin
      colour := 0;
      for plane := 1 to 4 do
        begin
          colour := colour shl 1 + (dots[plane] AND 1);
          dots[plane] := dots[plane] shr 1;
        end;
      colours[bit] := colour;
    end;
  end; {GetColourByte}
*)
external; {$L GETCBYTE.OBJ}

procedure GetEpsonLine (column: EgaCol; {input column of bytes from the image}
  Collength: ColSize; {length of the column in bytes}
  LastBit: BitCounter; {the last valid bit in the column}
  var PLine: Epson8DotLine); {output codes for printer}
{
  Convert a single image byte column to printer code form:

  Each screen image byte (8 pixels) translates on the
  printer to a pattern composed of eight upright printer
  pin images (*), as follows:

  Above MidLeft***
      v
  Above Left***>*****<***Above Right
  Below Left***>*****<***Below Right
      ^
      *****Below MidRight

  Each pin image (*) is obtained by sending a single byte
  to the printer in graphics mode.

  Close up, this pattern is made up of dots (.) like this:

```

```

. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .

. . . .
x x x x
x x x x
. . . .
. . . .
. . . .
. . . .
. . . .

```

Each dot (.) is bit in a pin image code. The MSB of the code is the topmost dot.

Each screen pixel is represented by a group of dots such as that highlited with (x). So the entire pattern represents 8 screen pixels, which in fact originate from a single line in an eight-pixel wide column on the screen (eight pixels because one video memory byte stores info about eight horizontal adjacent pixels). In addition each pattern is made up of two layers, misaligned slightly so that the second layer fills in the spaces of the first. Thus if the second layer looks like this,

```

. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .
. . . .

```

```

. . . .
. . . .
Y Y Y Y
Y Y Y Y
. . . .
. . . .
. . . .
. . . .

```

then we see that the highlighted pixel is actually represented on the printer as follows:

```

x x x x
Y Y Y Y
x x x x
Y Y Y Y

```

The EpsonGreyScale table stores the patterns for a single pixel in the following format:

```

LAYER:      1      2
          ===      ===

```

COLOUR n: ((X,X,X,X),(Y,Y,Y,Y)),

where n goes from 0 to 15. X is a number between 0 and 3 that represents the binary combination of part of a pixel, ie:

```

      x      0      0      1      1
X = x --> 0 = 0  1 = 1  2 = 0  and 3 = 1

```

Similarly Y is the code for the dots that originate in the secon layer.

Thus if we wished a pixel of colour 9 to print as a tiny oblong box, we might program:

```

          *****> 0 0 0 0
          *****> 1 0 0 1  = 1 0 0 1 for layer 1
0 0 0 0 *****
1 1 1 1 *****> 1 1 1 1
1 0 0 1 *****  ***> 1 1 1 1  = 3 3 3 3 for layer 2
1 1 1 1 *****

```

And so: COLOUR 9: ((1,0,0,1),(3,3,3,3)),

```

const
  EpsonGreyScale: array[ColourIndex,Layer1..Layer2,Left..Right] of 0..3 =
  (
    {
      LAYER:      1      2
      ===      ===
    }
    {COLOUR  0} ((0,0,0,0),(0,0,0,0)),
    {          1} ((0,0,0,0),(1,0,0,0)),
    {          2} ((0,0,0,0),(1,1,0,0)),
    {          3} ((0,0,0,0),(1,1,1,0)),
    {          4} ((0,0,0,0),(1,1,1,1)),
    {          5} ((1,0,0,0),(1,1,1,1)),
    {          6} ((1,1,0,0),(1,1,1,1)),
    {          7} ((1,1,1,0),(1,1,1,1)),
    {          8} ((1,1,1,1),(1,1,1,1)),
    {          9} ((1,1,1,1),(3,1,1,1)),
    {         10} ((1,1,1,1),(3,3,1,1)),
    {         11} ((1,1,1,1),(3,3,3,1)),
    {         12} ((1,1,1,1),(3,3,3,3)),
    {         13} ((3,1,1,1),(3,3,3,3)),
    {         14} ((3,3,1,1),(3,3,3,3)),
    {         15} ((3,3,3,3),(3,3,3,3))
  );

```

```

{}procedure GetEpson8Dot (colours: ColourByte;      {input image byte colours}
{}                        var codes: Epson8Dot);      {output print codes}
{}
{} Convert the colour info from
{} () a single byte from the image
{} () to a unit composed of dither
{} () pattern dots in printer pin
{} () image code form
{}
{}const

```

```

{} blanks: Epson8Dot = ((0,0), (0,0), (0,0), (0,0)), ((0,0), (0,0), (0,0), (0,0)));
{}
{}var
{} code: byte;                {temporary holder for a printer pin image code}
{} codeside: sidetype;        {determines which printer code is being done}
{} codeheight: heighttype;    { "-" }
{} codelayer: layertype;      { "-" }
{} bit: BitCounter;           {counts the bits of the screen image byte}
{} Moves: 0..3;               {used in aligning the basic Greyscale data}
{} n:0..3;
{}
{}begin
{} codes := blanks;
{} for codelayer := Layer1 to Layer2 do
{} begin
{}   codeheight := Above;
{}   for bit := 7 downto LastBit do
{}   begin
{}     Moves := bit mod 4;
{}     for codeside := Left to Right do
{}     begin
{}       code := EpsonGreyScale [GreyPalette[colours[bit]],codelayer, codeside];
{}       for n := 1 to Moves do code := code shl 2;
{}       if bit = 3 then codeheight := Below;
{}       codes [codelayer, codeside, codeheight] :=
{}         codes [codelayer, codeside, codeheight] OR code;
{}     end;
{}   end;
{} end;
{}end; {procedure GetEpson8Dot}

var
n,m: ColSize;
i: 1..4;
colours: ColourByte;
endreached: boolean;

begin
m := 1;
n := ColLength;
endreached := FALSE;
repeat
  GetColourByte (column[n],colours);
  GetEpson8Dot (colours, PLine[m]);
  if n = 1 then endreached := TRUE
  else begin
    Dec (n);
    Inc (m);
  end;
until endreached;
end; {procedure GetEpsonLine}

procedure EpsonPrint (PLine: Epson8DotLine;
  ColumnLength: word);

{
  Output the bit patterns in the
  "PLine" variable to an Epson
  type dot matrix printer
}

{}procedure Set_Home_Head;
{}begin
{} write (DumpDest, #13);
{}end;

{}procedure New_Line;
{}begin
{} write (DumpDest, #27+'J', char(23));
{}end;

{}procedure Send_single_density (No_of_codes: word);
{}begin
{} write (DumpDest, #27+'K', char(Lo(No_of_Codes)), char(Hi(No_of_Codes)));
{}end;

{}procedure Send_double_density (No_of_codes: word);
{}begin
{} write (DumpDest, #27+'L', char(Lo(No_of_Codes)), char(Hi (No_of_Codes)));
{}end;

{}procedure Send_quadruple_density (No_of_codes: word);
{}begin
{} write (DumpDest, #27+'Z', char(Lo(No_of_Codes)), char(Hi (No_of_Codes)));
{}end;

```

```

()procedure New_Layer;
()begin
() write (DumpDest, #27+'J', char(1));
()end;

var
dotcounter: word;
layer: LayerType;
side: SideType;
height: HeightType;
strikes: 1..10;
n: 1..10;

begin
for height := Above to Below do
begin
New_Line;
for layer := Layer1 to Layer2 do
begin
Set_home_head;
If layer = Layer2 then New_layer;
Case PrintScale of
Squashed: Send_quadruple_density (4*ColumnLength);
Normal: Send_double_density (4*ColumnLength);
Stretched: Send_single_density (4*ColumnLength);
end;
for DotCounter := 1 to ColumnLength do
for side := Left to Right do
write (DumpDest, char(PLine [dotcounter, layer, side, height]));
end;
end;
end; {procedure EpsonPrint}

```

```

procedure GetLJ2Line (column: EgaCol;
Collength: ColSize;
var PLine: LJ2DotLine);

```

```

{
Convert a single image byte column
to printer code form.

```

The screen is arbitrarily divided into pixel pairs - we use column-wise processing due to practical limitations (see last paragraph) so the pairs are actually vertical. But it is easier to imagine horizontal left-right pairs until the explanation in the last paragraph. Each pixel pair on the screen translates on the printer to a 4-byte column:

```

--
two pixels: --
--
--

```

Each "--" represents a single byte. A column of bytes contains the bit patterns for two pixels on the screen. The left nibbles encode the leftmost pixel of the pair, and the right nibble the rightmost pixel:

```

-          XXXX
left pixel: - equals XXXX
-          XXXX
-          XXXX

-          ZZZZ
right pixel: - equals ZZZZ
-          ZZZZ
-          ZZZZ

```

combined makes a four-byte column:

```

--          XXXXZZZZ
two pixels: -- equals XXXXZZZZ
--          XXXXZZZZ
--          XXXXZZZZ

```

Each "X" represents a '1' or a '0' bit in a left (Most Significant) nibble, and each "Z" a right nibble in the printed byte.

The colour of each pixel is used as the index into the LJ2GreyScale table.

```
8-pixel wide      32 printed bytes - a unit in the
screen column     LJDotLine array:
byte pair:
```

Bytes are sent to the printer row-wise - the rows are numbered. To reconcile this with the preceding discussion note that the paper is imagined "sideways" - the result is a landscape orientation, but it was processed portrait (row) wise:

[illegible]

PRTEGA.PAS B83

```

{}begin
{} codes := LJ2GreyScale[GreyPalette[rightcolour]];
{} for i := 1 to 4 do
{} codes[i] := codes[i] OR (LJ2GreyScale[GreyPalette[leftcolour],i] shl 4);
{}end; {GetLJ2DotPair}

var
m: ColSize;
n: 0..MAXCOLPIX;
i: 1..4;
bit: BitCounter;
leftcolours, rightcolours: ColourByte;

begin
m := 1;
n := ColLength;
Repeat
GetColourByte (column[n], leftcolours);
Dec(n);
If PrintScale <> Stretched
then begin
GetColourByte (column[n], rightcolours);
Dec(n);
end
else rightcolours := leftcolours;

for bit := 0 to 7 do
GetLJ2DotPair (leftcolours[bit], rightcolours[bit], PLine[m,bit]);
Inc (m);
Until(n < 2);
If n <> 0 then
begin
GetColourByte (column[n], leftcolours);
for bit := 0 to 7 do
GetLJ2DotPair (leftcolours[bit], 0, PLine[m,bit]);
end;

end; {procedure GetLJ2Line}

procedure LJ2Setup;
{
Set the laser printer resolution
to 300 dpi, set portrait mode and
enter graphics mode with the cursor
at the left margin.
}

begin
write(DumpDest, #27, '*t300R');
write(DumpDest, #27, '&l00');
write(DumpDest, #27, '*r0A');
end; {LJ2Setup}

procedure LJ2ShutDown;
{
Exit graphics mode on the laser
printer, reset the printer and
form feed.
}

begin
write(DumpDest, #27, '*rB');
write(DumpDest, #27, 'E');
end; {LJ2ShutDown}

procedure LJ2Print (PLine: LJ2DotLine;
ColumnLength: word;
LastBit: BitCounter);
{
Output the bit patterns in the
"PLine" variable to a HP
Laserjet II type laser printer
}

{}procedure NewLine(length:word);
{}var l: string;
{}begin
{} str (length, l);
{} write (DumpDest, #27, '*b', l, 'W');
{}end;

```

```

var
  i: word;
  bit: BitCounter;
  row: 0..4;

begin
  If PrintScale <> Stretched then ColumnLength := ColumnLength div 2 + 1;
  for bit := 7 downto LastBit do
    for row := 1 to 4 do
      begin
        NewLine(ColumnLength);
        for i := 1 to ColumnLength do write (DumpDest, char(PLine[i,bit,row]));
      end;
    end;
  end; {LJ2Print}

procedure PSPrintCol (column: EgaCol; {input column of bytes from the image}
                     ColLength: ColSize); {length of the column in bytes}
{
  Convert a single image column to PostScript image data and send it.
  The column must be sent as a raster scan, so the image is
  printed on its side - the msb's of all the column's bytes
  make up one "row" of pixels on the paper. No printer dot
  pattern is required as for the other printer types, Postscript accepts
  a single grey shade value for each pixel. In this each grey shade
  value takes 4 bits, so it fits neatly into a hex digit, which makes it
  easy to output each pixel separately. Only thing to worry about is that
  the number of hex digits must be even or else the extra digit will join
  to the next hex number and skew things awfully. Unfortunately neither
  padding nor leaving the column number as it was seemed to work here,
  possibly due to other parameters not being set up right in the PS file.
  For the system to work, then, only only even-width pictures should
  be dumped.
}

var
  n: ColSize;
  i: 0..7;
  c: 0..81;
  colourcol: array[1..MAXCOLPIX] of ColourByte;

const
  hextable: array[ColourIndex] of char =
    ('0','1','2','3','4','5','6','7','8','9','A','B','C','D','E','F');
begin
  c := 0;
  for n := Collength downto 1 do GetColourByte (column[n],colourcol[n]);
  (*
  if odd (Collength) then
    begin
      dec(Collength);
      fillchar (colourcol[Collength],sizeof(colourcol[Collength]),#0);
      {assume padding colour is 0 - not really good assumption, but what
      can you do? Put in a special index colour, say 16, which is always
      mapped to 0 in GreyPalette, so will always be white (black) on
      paper. But we'll leave it for now.}
    end;
  *)
  for i := 7 downto 0 do
    begin
      for n := Collength downto 1 do
        begin
          write (DumpDest, hextable[GreyPalette[colourcol[n,i]]);
          inc(c);
          if c = 70 then
            begin
              writeln (DumpDest);
              c := 0;
            end;
          end;
        end;
      end;
    end;
  end; {procedure PSPrintCol}

procedure PSSetup(width, height: word);
{
  The postscript that comes before the data - basically the 'image' statement,
  but set out for easy reading and modification later. The passed in
  parameters are the width and height in pixels of the image.
}
begin
  writeln(DumpDest,'%!PS-Adobe-2.0 EPSF-1.0');
  writeln(DumpDest,'%Creator: PrtEga');
  writeln(DumpDest,'%BoundingBox: 0 0 ',width/2*2.835,' ', height/2*2.835);
  writeln(DumpDest,'%EndComments');
  writeln(DumpDest);

```

```

writeln(DumpDest,'save');
writeln(DumpDest,'/mm {2.835 mul} def');
writeln(DumpDest,'/pixelsperline ',width,' def');
writeln(DumpDest,'/totallines ',height,' def');
writeln(DumpDest,'/bitsperpixel 4 def');
writeln(DumpDest,'/Xbottomleft 0 mm def');
writeln(DumpDest,'/Ybottomleft 0 mm def');
writeln(DumpDest,'/widthpic ',width/2:0:4,' mm def');
writeln(DumpDest,'/heightpic ',height/2:0:4,' mm def');

writeln(DumpDest,'/workstring pixelsperline string def');
writeln(DumpDest,'/getimage (');
writeln(DumpDest,'pixelsperline totallines bitsperpixel');
writeln(DumpDest,'[pixelsperline 0 0 totallines neg 0 totallines]');

writeln(DumpDest,'{currentfile workstring readhexstring pop}');
writeln(DumpDest,'image ) def');
writeln(DumpDest,'gsave');
writeln(DumpDest,'[1 exch sub] settransfer % negate image');
{
EXAMPLE: changing the halftone frequency from default (60) to 75
writeln ('currentscreen 3 -1 roll pop 75 3 1 roll setscreen');
}
writeln(DumpDest,'Xbottomleft widthpic add Ybottomleft heightpic add translate');
writeln(DumpDest,'widthpic neg heightpic neg scale');
writeln(DumpDest,'getimage');
end;

```

procedure PSShutdown;

```

begin
  writeln(DumpDest);
  writeln(DumpDest,'grestore');
  writeln(DumpDest,'restore');
  writeln(DumpDest,'showpage');
end;

```

procedure ChooseGreyPalette(choice: GreyPaletteChoice);

```

{
Chooses values for GreyPalette,
which maps colour index numbers
to GreyScale table entries. The values
are chosen intelligently by looking at
the current palette. Each palette colour
is assigned an effective brightness
level. This is done by weighting the
primary colours to compensate for the eye
sensitivity of the human eye. Note that
these weighting values are subjective and
crude, and could be better estimated using
rigorous theoretical considerations.

```

Typically, the above will result in some colours being mapped to the same greyness level. Before assigning the GreyPalette values therefore, the contrast between such colours is used to allocate different grey scales.

Note that the colours as returned by GetPalette are coded as a six bit binary number, and the bits have the following meaning:

MSB --> RGBrgb <-- LSB

where the brightness levels for red are

```

rR = 00 --> brightness level 0 (red off);
rR = 01 --> brightness level 1 (dim red);
rR = 10 --> brightness level 2 (red);
rR = 11 --> brightness level 3 (light/bright red);

```

and similarly for blue and green.

```

const B = 10;           {Blue often appears dimmest}
      R = 12;           {Red appears marginally brighter than blue}
      G = 20;           {Green is significantly brighter than blue}
      MaxBright = 3*(B+R+G);

```

```

type UnquantizedGreyLevel = 0..MaxBright;
     QuantizedGreyIndex = 0..15;

```

```

var Pal: PaletteType;
ScratchGreyPalette: array [ColourIndex] of
  record
    greyness: UnquantizedGreyLevel;
    usedflag: boolean;
  end;
GreyGroupings: array [QuantizedGreyIndex] of
  record
    colour: ColourIndex;
    greylvl: QuantizedGreyIndex;
  end;
colournum: ColourIndex;
i, greylevel, StartGroup, EndGroup: QuantizedGreyIndex;
bigcolour: shortint;
finished, endreached, levelfound: boolean;

()function Quant16 (gl: UnquantizedGreyLevel): QuantizedGreyIndex;
()
() returns the grey scale obtained by
() quantising the input to 16 levels.
()
()begin
() if gl = MaxBright then Quant16 := 15
() else Quant16 := Trunc(gl/MaxBright*16);
()end; {}function Quant16}

()procedure Sort (startpt, endpt: ColourIndex);
()
() sorts values in 'GreyGroupings'
() according to the corresponding grey
() levels in ScratchGreyPalette (which
() are unquantised). The sorting starts
() with the entry at the "start" position
() in GreyGroupings and ends with
() the "end" position. It is a bubblesort.
()
()
()var
() NoSwaps: boolean;
() i: QuantizedGreyIndex;
() temp: ColourIndex;
()
()begin
() repeat
() NoSwaps := TRUE;
() for i := startpt+1 to endpt do
() begin
() if ScratchGreyPalette[GreyGroupings[i].colour].greyness <
() ScratchGreyPalette[GreyGroupings[i-1].colour].greyness
() then begin
() temp := GreyGroupings[i].colour;
() GreyGroupings[i].colour :=
() GreyGroupings[i-1].colour;
() GreyGroupings[i-1].colour := temp;
() NoSwaps := FALSE;
() end;
() end;
() until (Noswaps);
()end; {}procedure Sort}

begin

  {Calculate grey level value for each colour}

  GetPalette(Pal);
  for colournum := 0 to 15 do
  with ScratchGreyPalette[colournum] do
  begin
    bigcolour := Pal.colors[colournum];
    greyness := ((bigcolour shr 0) AND 1)*B*2 {blue msb}
      +((bigcolour shr 1) AND 1)*G*2 {green msb}
      +((bigcolour shr 2) AND 1)*R*2 {red msb}
      +((bigcolour shr 3) AND 1)*B {blue lsb}
      +((bigcolour shr 4) AND 1)*G {green lsb}
      +((bigcolour shr 5) AND 1)*R; {red lsb}
    usedflag := FALSE;
  end;

```

{ScratchGreyPalette now contains the values which, when quantized, ought to go into GreyPalette. However, some colours may have the same grey scale (either now or after quantization and these could be separated and assigned the unused grey scale values. The advantage of this is that colours of the same effective brightness which may still be perfectly distinguishable will be have different greyscales - in fact every available colour will have a different grey scale. This can be a disadvantage if there are colours that map to the identical colour in the colour palette. These should theoretically have the identical grey scale. Another disadvantage is that if there are many colours with the the same effective brightness, they will cause a distortion in the overall grey scale assignments because 'space' in the grey palette will have to be found for them, 'pushing' legitimate assignments to artificial positions.

Because of the above tradeoff, the choice has been left to the user as to whether or not to 'expand' the grey palette or to leave all colours in their original quantised assignments. The input parameter to this procedure, 'choice', should be 'AsQuantized' to leave quantized values, 'Differentiated' to expand the assignments to fit the available grey scale range.

The following code groups the colours that map to the same grey scale together, in preparation for differentiating them. The 'AsQuantized' option does however use the results (greylvl values) of this stage, although it needn't have.

```

}
greylevel := 0;
i := 0;
finished := FALSE;

repeat
  colournum := 0;
  levelfound := FALSE;
  endreached := FALSE;

  repeat
    if (ScratchGreyPalette[colournum].usedflag = FALSE) and
      (Quant16(ScratchGreyPalette[colournum].greyness) = greylevel)
    then levelfound := TRUE
    else if colournum = 15 then endreached := TRUE
        else Inc (colournum);
  until levelfound or endreached;

  if not levelfound
    then Inc (greylevel)
    else begin
      GreyGroupings[i].colour := colournum;
      GreyGroupings[i].greylvl := greylevel;
      ScratchGreyPalette[colournum].usedflag := TRUE;
      If i = 15 then finished := TRUE
        else Inc (i);
    end;
until finished;

```

{Differentiating the colours:

Let's say that the colours "Dark Green" in the '3' position of the palette, "Bright Red" in the '8' position, and "Medium Yellow" in the '12' position map to the same quantized grey scale, say 5. The contents of GreyGroupings after the above operations may be:

```

position 0 -->      (
                    (first entry),
                    .
                    .
                    .
position 9 -->      (colour: 3; greylvl:5),
position 10 -->     (colour: 8; greylvl:5),
position 11 -->     (colour:12; greylvl:5),
                    .
                    .
position 15 -->     (last entry)
                    );

```

The order that they appear is determined by their order of appearance in the original palette, but we should rather make the order (which represents the grey scale assignment, where 0 is darkest and 15 lightest) correspond in some way to the colour characteristics. The method below uses the unquantized grey level(intensity) information to order the colours within their 9-10-11 grouping, but another possible method would be to use rule-of-thumb knowledge of the contrast characteristics of the colours. Thus if it is known that Dark Green contrasts with Bright Red more than it does with Medium Yellow, the order should be '3, 12, 8' rather than '3, 8, 12' as above.)

{Order the adjacent-groups according to original intensity}

```
StartGroup := 0;
for EndGroup := 1 to 15 do
  If GreyGroupings[EndGroup].greylvl <>
    GreyGroupings[StartGroup].greylvl
  then begin
    if (StartGroup <> EndGroup-1) then Sort (StartGroup, EndGroup-1);
    StartGroup := Endgroup
  end
  else if EndGroup = 15 then Sort (StartGroup,15);
```

{Finally, transfer values to GreyPalette};

```
case choice of
  ASQuantized: for i := 0 to 15 do with GreyGroupings[i] do
    GreyPalette[colour] := greylvl;
  Differentiated: for i := 0 to 15 do with GreyGroupings[i] do
    GreyPalette[colour] := i;
end; {case}
```

end; {procedure ChooseGreyPalette}

```
procedure EgaDump (x1,y1,x2,y2: word);
{
  (x1,y1) and (x2,y2) define respectively
  the top left corner and bottom right
  corner of a rectangular region on the
  (EGA) screen display. This is dumped to
  the printer.
}
```

```
var
  imagesiz: word;
  image: pointer;
  screencolumn, columnsave: ^EgaImageCol;
  Collength: word;
  colcorner: word;
  EpsonLine: ^Epson8DotLine;
  LJ2Line: ^LJ2DotLine;
  colcounter, lastcol: integer;
  endbit: BitCounter;
  txsave: textsettingstype;
  xsave, ysave: word;
```

```
begin
  New (screencolumn);
  New (columnsave);

  Case Printer of
    Epson:
      begin
        New (EpsonLine);
      end;
    LaserJet2:
      begin
        LJ2SetUp;
        New (LJ2Line);
      end;
    Postscript:
      begin
        if odd(y2-y1+1) then dec(y2); {kludge - adjust to even column length}
          {because odd ones didn't work}
        PSSetup(y2-y1+1, ((x2-x1) div 8 +1)*8); {swap parameters because}
          {image is on its side}
      end;
  end; {case}
```

```

colcorner := x1;
lastcol := (x2-x1) div 8+1;
endbit := 0;
GetImage (0,0, 7,GetMaxY, columnsave^);
gettextsettings (txsave);
xsave := getx;
ysave := gety;
settextstyle (DefaultFont, VertDir, 1);
settextjustify (RightText,BottomText);

for colcounter := 1 to lastcol do
begin
  GetImage (colcorner,y1, colcorner+7,y2, screencolumn^);
  PutImage (0,0, screencolumn^, CopyPut);
  outtextxy(8,getmaxy,'PrintEga by IC');
  ColLength := screencolumn^.length+1;
  if colcounter = lastcol then endbit := (8-((x2-x1+1) mod 8)) mod 8;

  Case Printer of
    Epson:
      begin
        GetEpsonLine (screencolumn^.pixels,ColLength, endbit, EpsonLine^);
        EpsonPrint (EpsonLine^,ColLength);
      end;
    LaserJet2:
      begin
        GetLJ2Line (screencolumn^.pixels,ColLength, LJ2Line^);
        LJ2Print (LJ2Line^, ColLength, endbit);
      end;
    Postscript: PSPrintCol(screencolumn^.pixels, Collength);
  End; {case}

  colcorner := colcorner + 8;
end;

PutImage (0,0, columnsave^, CopyPut);
settextstyle (txsave.Font, txsave.Direction, txsave.CharSize);
settextjustify (txsave.Horiz, txsave.Vert);
moveto (xsave, ysave);

Case Printer of
  Epson:
    begin
      Dispose (EpsonLine);
    end;
  LaserJet2:
    begin
      LJ2Shutdown;
      Dispose (LJ2Line);
    end;
  Postscript:
    begin
      PSShutdown;
    end;
end; {case}

Dispose (columnsave);
Dispose (screencolumn);

end; {EgaDump}

procedure InvertGreyPalette;
{
  Calling this will invert the grey
  scale info in GreyPalette. A Colour
  which previously mapped to Black will
  now map to White, etc.
}
var i: ColourIndex;
Begin
  for i := 0 to 15 do GreyPalette[i] := 15-GreyPalette[i];
End; {Procedure InvertGreyPalette}

procedure PrintEga (DumpDestStr:string;
  PrinterPar:PrinterType;
  PrintScalePar:PrintScaleType;
  PrintFormPar:PrintFormType;
  x1,y1, x2,y2: integer);
var
  escaped: boolean;
  response: string[1];
begin
  Printer := PrinterPar;
  PrintScale := PrintScalePar;
  if Printer = LaserJet2

```

```

    then if PrintScalePar = Squashed then PrintScale := Normal;
PrintForm := PrintFormPar;
Assign (Dumpdest, DumpDestStr);

(* This is removed when you don't want to use ICON, MOUSE
repeat
{$I-}
    response := ' ';
    rewrite (DumpDest);
    write (DumpDest, #0);
    If IOresult <> 0 then
    begin
        response := Prompt (200,175,'Printer or file error: ' +
            'A - abort, ANY KEY - retry print','',1,escaped);
        response := upcase(response[1]);
        if response = 'A'
        then begin
            close (dumpdest); {in case the error was on the write}
            escaped := (IOresult = 0); {use up IOresult}
            exit; {leave Printega}
        end
        else response := 'R';
    End{If};
    {$I+}
    until response = ' ';
*)

rewrite (DumpDest);
if PrintForm = Split
then Case Printer of
    Epson: begin
        EgaDump (x1,y1,x2,(y2-y1) div 2);
        writeln (DumpDest, #27,#12); {form feed}
        EgaDump (x1,(y2-y1) div 2+1, x2,y2);
        end;
    LaserJet2: begin
        EgaDump (x1,y1,(x2-x1) div 2,y2);
        EgaDump ((x2-x1) div 2+1,y1,x2,y2);
        end;
    End{Case}
else EgaDump (x1,y1,x2,y2);

close (DumpDest);
End{Procedure};

{INITIALIZATION}

var i: integer;
begin
    for i := 0 to 15 do GreyPalette[i] := i;
    {This is the default GreyPalette.}
    {Note that if the colour palette }
    {is changed at any stage, then }
    {'ChooseGreyPalette' should be }
    {called afterwards (unless you }
    {are setting 'GreyPalette' your- }
    {self. }
end.

```

B1.12.1 GETCBYTE.ASM

```
;Procedure GetColourByte (dots:EgaByte, colours:PtrColourByte);
;in PRTEGA.PAS

; Procedure to obtain the colour index (0 to 15)
; of the pixels in a byte on the screen (actually
; four bytes -- "overlaid" on top of each other
; in planes. The first plane gives the LSB of the
; colour index number).
;
; Input is the 'dots' the four bytes from the EGA screen,
; Output is the eight colour values of the pixels, stored at the
; address given by 'colours'

.MODEL TPASCAL
.CODE
IDEAL
PROC GetColourByte NEAR dots:DWORD, colours:FAR PTR BYTE ;colours is an array
                                                    ;of bytes
PUBLIC GetColourByte
    mov bx, [WORD dots] ;store all the 'dots' elements in the bl,bh
    mov dx, [WORD dots+2] ;dl and dh registers, for speed.
;    mov ax,ds
;    mov es,ax
;    mov ax,[colours]
;    mov di,ax
    les di, [colours] ;es:di points to the colour values storage space
    xor ah,ah ;ah stays 0 for the entire routine
    mov cx,8 ;do the loop 8 times, 1 for each pixel
PixelLoop:
    xor al,al ;clear the colour value

    shr bl,1 ;shift first plane byte LSB into Carry
    rcl al,1 ;shift Carry into colour LSB

    shr bh,1 ;shift second plane byte LSB into Carry
    rcl al,1 ;shift Carry into colour LSB

    shr dl,1 ;shift third plane byte LSB into Carry
    rcl al,1 ;shift Carry into colour LSB

    shr dh,1 ;shift fourth byte LSB into Carry
    rcl al,1 ;shift Carry into colour LSB

    stosb ;store the colour value at ds:si, inc si.
    loop PixelLoop

    mov ax,@data
    mov ds,ax
    ret
ENDP GetColourByte
END
```

B1.13 DRAWER.PAS

```
UNIT DRAWER;
INTERFACE
USES BCRTYPES;

{ $define EGALO} {Enable this if using dual-frequency monochrome monitor}
{attached to EGA card, to get realistic sonagram view}

Procedure OutInvertText(txt: string);
{
  Like OutText but prints the string inverted about its centre point.
}

Procedure Display (CallName, SpecName, SonaName, PeakName,
                  ClassName, ElementName: NameStr);
{
  Executes DRAWER process.
  Graphical menu-based environment for viewing and printing files in a BCR
  experiment system.
}

IMPLEMENTATION
USES Dos, Crt, Graph, BCDMANIP, ICON, PRTEGA, GENLIB, BACKGND;

TYPE
  realtoseg = record a,b: segmenttype end;

VAR
  Sonagram_storage: file of segmenttype;
  Peak_frames: file of peakframe;
  Classified_frames: file of classifiedframe;
  Elements: file of elementtype;
  driver: integer;
  GlobalCallName, GlobalSpecName: NameStr;

CONST
  XORIGIN = 39;
  YORIGIN = 30; {absolute coords of the sonagram origin}

VAR
  LINESPERPAGE: word;
  {Number of horizontal lines per sonagram page. Gets its value in the
  initialization section, depending on graphics hardware}

  x1box,y1box,x2box,y2box: integer; {coordinates of the default box used by
  'ChooseBox'}

Function ClassColour (c:classtype): byte;
{
  Returns a colour for each class;
}
begin
  ClassColour := round(ord(c)*14/5+1);
End;

Procedure OutInvertText (txt: string);

var
  textinfo: textsettingstype;
  viewinfo: viewporttype;
  oldx, oldy: integer;
  HBox, VBox: integer;

{}Procedure CalculateTextBox(Dir: integer; Txt: string;
{}                          var XBox,YBox: integer);
{}
{}
{} calculates the horizontal (XBox) and vertical (YBox) dimensions of an
{} imaginary box surrounding the text (Txt) based on the current writing
{} direction (Dir).
{}
{}begin
{}  Case Dir of
{}    HorizDir:begin
{}      XBox := textwidth(txt);
{}      YBox := textheight(txt);
{}    end;
{}    VertDir: begin
{}      XBox := textheight(txt)+1; {bug?}
{}      YBox := textwidth(txt);
{}    end;
```

```

{}      end;
{}      End; {Case}
{}End; {Procedure CalculateTextBox}

{}Procedure SetViewportToTextBox(x,y,XBox, YBox, HorizJust, VertJust,
{}                               Dir, Font: integer);
{}
{}
{} sets the viewport to a box given by XBox and YBox, as if it was outtexted
{} at (x,y) based on the current justification and direction (HorizJust,
{} VertJust and Dir).
{}
{}var
{}  Xdisp, Ydisp: integer;
{}begin
{}  Case HorizJust of
{}    LeftText: if not ((Dir = VertDir) and (Font = DefaultFont))
{}               then xdisp := 0
{}               else xdisp := XBox; {This IF is needed because of a bug -}
{}    CenterText: xdisp := XBox div 2;
{}    RightText: xdisp := XBox;
{}  End; {Case}
{}
{}  Case VertJust of
{}    TopText: ydisp := 0;
{}    CenterText: ydisp := YBox div 2;
{}    BottomText: ydisp := YBox;
{}  End; {Case}
{}
{}  setviewport(x-Xdisp, y-Ydisp,
{}              x-Xdisp+Xbox, y-Ydisp+YBox, ClipOn);
{}
{}End; {Procedure SetViewportToTextBox}

{}Procedure InvertBox (Xbox, YBox: integer);
{}
{} Inverts a box of size Xbox,Ybox positioned at 0,0 in the current viewport
{}
{}var
{}  pixel: word;
{}  xp,yp: integer;
{}begin
{}  For yp := 0 to VBox div 2-1 do
{}    begin
{}      For xp := 0 to HBox-1 do
{}        begin
{}          pixel := getpixel(xp,yp);
{}          putpixel(xp,yp,getpixel(Xbox-xp-1,YBox-yp-1));
{}          putpixel(Xbox-xp-1,Ybox-yp-1,pixel);
{}        End{For};
{}      End{For};
{}    End{For};
{}End; {Procedure InvertBox}

begin
  oldx := getx; oldy := gety;
  gettextsettings(textinfo);
  getviewsettings (viewinfo);

  CalculateTextBox(textinfo.Direction, Txt, Hbox, VBox);

  {first invert the box}
  SetViewportToTextBox(oldx,oldy,HBox,VBox,textinfo.Horiz, textinfo.Vert,
                      textinfo.Direction, textinfo.Font);
  InvertBox(Hbox, VBox);
  with viewinfo do setviewport(x1,y1,x2,y2,clip);

  {now place text in box}
  outtextxy(oldx,oldy,Txt);

  {now invert the box back, leaving the text inverted}
  SetViewportToTextBox(oldx,oldy,HBox,VBox,textinfo.Horiz, textinfo.Vert,
                      textinfo.Direction, textinfo.Font);
  InvertBox(Hbox, VBox);
  with viewinfo do setviewport(x1,y1,x2,y2,clip);

  if (textinfo.Direction = HorizDir) and (textinfo.Horiz = LeftText)
  then moveto (oldx+HBox, oldy) {only advance the pointer in this case}
  else moveto (oldx, oldy);
End; {Procedure InvertText}

{$F+}      {Procedures to be called via procedure-type variables}

```



```

        Else decentname := TRUE;
    End{If};
    Until escaped or decentname or (FileName = '');

    if not escaped and (FileName <> '')
    then PrintEga (FileName,printer,printscale, printform,
        xlbox,ylbox,x2box,y2box);

    MenuQuitFlag := TRUE;
    End{If};
End{Procedure DumpToFile};

Procedure ScreenDump;
const
    dumpstr: string[10] = 'Begin Dump';
begin
    Menu (500,57,4,
        @dumpstr,      DumpToFile,
        @printerstr,   TogglePrinter,
        @printscalestr, TogglePrintScale,
        @printformstr, TogglePrintForm,
        nil,null,
        nil,null,
        nil,null,
        nil,null,
        nil,null,
        nil,null);
End{Procedure ScreenDump};

VAR
    fAxis, tAxis, EndfAxis, EndtAxis:integer;
    totallines:longint;
    LinesPerMsec, PointsPerKHz: real;
    {NOTE the above two variables must be reals. If they are changed (eg
    to doubles),the definition of the 'realtoseg' type cast must be changed,
    as well as the type written to Sonagram_storage by SONAGRAM.}

CONST
    timeaxisstr: string[12] = 'Time AxisABS';
    TimeAxisAbs: boolean = TRUE; {enables Absolute scale on time axis}

Procedure ToggleTimeAxisAbs;
begin
    TimeAxisAbs := not TimeAxisAbs;
    if TimeAxisAbs then timeaxisstr := 'Time AxisABS'
        else timeaxisstr := 'Time AxisREL';
End{Procedure ToggleTimeAxisAbs};

CONST
    freqaxisstr: string[12] = 'Freq AxisON';
    FreqAxisOn: boolean = TRUE; {enables frequency axis drawing}

Procedure ToggleFreqAxisOn;
begin
    FreqAxisOn := not FreqAxisOn;
    if FreqAxisOn then freqaxisstr := 'Freq AxisON'
        else freqaxisstr := 'Freq AxisOFF';
End{Procedure ToggleFreqAxisOn};

VAR graphtitle: string[65];
Procedure DrawAxes;
var
    selected, escaped: boolean;
    unseenmarks: longint;
    textinfo: textsettingstype;

{}Procedure FreqAxis;
{}var
{}    axisdrawn,marklabel: real;
{}    firstdigit: string[1];
{}begin
{}    moveto (tAxis,fAxis);      {draw the axis line}
{}    lineto (EndfAxis,fAxis);
{}
{}    settxtjustify (centertext, bottomtext);
{}
{}    unseenmarks := -1;      {This and the 'if' ensure the zero label is drawn}
{}    if tAxis <> XORIGIN then unseenmarks := trunc((tAxis-XORIGIN)/PointsPerKHz);
{}    axisdrawn := (unseenmarks+1)*PointsPerKHz - (tAxis-XORIGIN);

```

```

{}
{} While axisdrawn <= EndfAxis - tAxis do      {ticks and labels}
{} begin
{}   moveto (tAxis,fAxis);                      {axis tick}
{}   moverel (round(axisdrawn),0);
{}   linerel (0,-3);
{}
{}   moverel (0,-2);                            {axis tick label}
{}   marklabel := (tAxis-XORIGIN+axisdrawn)/PointsPerKHz;
{}   OutInvertText(strung(1,1,marklabel));
{}
{}
{}   firstdigit := copy (strung(4,2,marklabel),1,1);
{}   outtext (firstdigit);
{}   moverel (-1,0); OutInvertText(firstdigit);      {go over it for boldness}
{}   moverel (1+textheight(firstdigit) div 2,-textwidth (firstdigit));
{}   linerel (0,-1); moverel (-1,0); linerel (0,1);  {decimal point}
{}   moverel (1-integer(textheight(firstdigit)) div 2,-3);
{}   outtext (copy (strung(4,2,marklabel),3,2));
{}   moverel (-1,0); OutInvertText (copy (strung(4,2,marklabel),3,2)); {go over it for
boldness}
*)
{}
{}   axisdrawn := axisdrawn+PointsPerKHz;
{} End;
{}
{}End{Procedure FreqAxis};

{}Procedure TimeAxis;
{}const MarkTime = 100; {number of milliseconds between axis marks}
{}var
{}  axisdrawn: real;
{}  freqpoints, previousmarks:longint;
{}
{}begin
{}  moveto (tAxis,faxis);                      {axis line}
{}  lineto (tAxis,EndtAxis);
{}
{}  axisdrawn := 0;
{}  If TimeAxisAbs then
{}  begin
{}    freqpoints := totallines-LINESPERPAGE+fAxis-YORIGIN;
{}    previousmarks := -1;                      {to ensure the zero label is drawn}
{}    if freqpoints <> 0                          {when freqpoints = 0}
{}    then previousmarks := trunc(freqpoints/LinesPerMsec/MarkTime);
{}    axisdrawn:= (previousmarks+1)*MarkTime*LinesPerMsec - freqpoints;
{}  End{If};
{}
{}  setttextjustify (righttext, centertext);
{}
{}  While axisdrawn <= (EndtAxis-fAxis) do
{}  begin
{}    moveto (tAxis,fAxis);                      {axis tick mark}
{}    moverel (0,round(axisdrawn));
{}    linerel (-4,0);
{}    linerel (-3,0);
{}
{}    moverel (-5,0);                            {axis tick label}
{}    if TimeAxisAbs
{}    then OutInvertText (Strung(4,2,(totallines-LINESPERPAGE+fAxis-
{}      YORIGIN+axisdrawn)/LinesPerMsec/1000))
{}    else OutInvertText (Strung(4,2,axisdrawn/LinesPerMsec/1000));
{}
{}    moverel (-1,0);                            {now do it again, for boldness}
{}    if TimeAxisAbs
{}    then OutInvertText (Strung(4,2,(totallines-LINESPERPAGE+fAxis-
{}      YORIGIN+axisdrawn)/LinesPerMsec/1000))
{}    else OutInvertText (Strung(4,.,axisdrawn/LinesPerMsec/1000));
{}
{}  axisdrawn := axisdrawn+MarkTime*LinesPerMsec;
{}  End;
{}End{Procedure TimeAxis};

begin
  ChooseBox (tAxis, fAxis, EndfAxis, EndtAxis, selected);
  If selected then
    begin
      if tAxis < XORIGIN then tAxis := XORIGIN;
      if fAxis < YORIGIN then fAxis := YORIGIN;
      if EndfAxis < XORIGIN then EndfAxis := XORIGIN;
      if EndtAxis < YORIGIN then EndtAxis := YORIGIN;
    end
  end

```

```

setcolor (white);
gettextsettings (textinfo);
setusercharsize (1,1,5,3);
settextstyle (defaultfont, vertdir,usercharsize);

Graphtitle := Prompt(400,80,'Plot Title?',Graphtitle,
                    (EndtAxis-fAxis) div textwidth('0'), escaped);

If not escaped then
begin
    settextjustify (righttext, centertext);
    moveto (EndfAxis, (EndtAxis+fAxis) div 2);
    OutInvertText (Graphtitle);
    (*
    moverel(-1,0);
    OutInvertText (Graphtitle);
    *)
    if FreqAxisOn then FreqAxis;
    TimeAxis;
end;

with textinfo do settextstyle (font, direction, charsize);
with textinfo do settextjustify (horiz, vert);

MenuQuitFlag := TRUE;
End{If};
End{Procedure DrawAxes};

Procedure AxisMenu;
const
    drawaxisstr: string[10] = 'Draw Axes';
begin
    Menu (500,65,3,
        @drawaxisstr, DrawAxes,
        @freqaxisstr, ToggleFreqAxisOn,
        @timeaxisstr, ToggleTimeAxisAbs,
        nil,null,
        nil,null,
        nil,null,
        nil,null,
        nil,null,
        nil,null,
        nil,null);
End{Procedure AxisMenu};

TYPE intervaledge = (Begint, Endint);

CONST
    Interval: array[Begint..Endint] of
        record
            line: longint;
            name: string[5];
        end
        = ((line: 100000; name: 'START'),
          (line: 100000; name: 'END'));

OverPlot: boolean = FALSE;
overplotstr: string[12] = 'Overplot OFF';

CONST
    MAXHALFPAGES = 100; {twice maximum number of sonagram pages expected}

VAR
    SSPosRecord: array[0..MAXHALFPAGES] of record
        pos:word;
        line:longint;
    end;
    {remembers the file positions of the starts of half-page intervals
    in the sonagram, and their corresponding line numbers}
    LastSSpos: 0..MAXHALFPAGES; {points to the last half page position}

Procedure DrawSonaPage;

{}Procedure RecordSSPos;
{}begin
{}    inc (LastSSPos);
{}    SSPosRecord[LastSSPos].pos := filepos(Sonagram_Storage);
{}    SSPosRecord[LastSSPos].line := totallines;
{}End{Procedure};

```

```

var
  lineno:word;
  segment: segmenttype;
begin
  if not OverPlot then cleardevice;
  lineno := 0;
  While (lineno < LINESPERPAGE) do
  begin
    If not eof (Sonagram_storage) then
    begin
      moveto (XORIGIN,YORIGIN+lineno);

      {draw a line}
      Repeat
        If not eof (Sonagram_storage) then
        begin
          read (Sonagram_storage,segment);
          if (totallines >= Interval[Begin].line) and
             (totallines <= Interval[Endint].line)
          then setcolor (15-(segment.level mod 16))
          else setcolor (segment.level mod 16);
          if segment.y <= maxint
          then lineto (segment.y+XORIGIN,YORIGIN+lineno)
          else runerror(UserMessage('Bad sonagram file'));
        End{If};
      Until segment.level >= 16;
    End{If};

    inc (lineno);
    inc (totallines);

    if totallines mod (LINESPERPAGE div 2) = 0 then RecordSSPos;
  End{While};
End{Procedure DrawSonaPage};

VAR
  MaxMag: real; {Maximum magnitude of peak in signal- BCDread by main procedure}
  WindowTime: real; {Length of a single frame in milliseconds}
  peaksexist: boolean; {Indicates whether the peak frame file exists}
  classesexist: boolean; {Indicates whether the classified frame file exists}
  elementsexist: boolean; {Indicates whether the elements file exists}

Procedure DrawPeakPage;
var
  currentframe: longint;
  lineno: word;
  peak: peakframe;
  pastframes: real;
  i: 1..PeaksPerFrame;
  ch: string[1];
  escaped: boolean;
{
  PlotAmplitude: boolean;
  magsum, magscale: real;
}
begin
  if peaksexist
  Then begin
    if not OverPlot then cleardevice;
    pastframes := SSPosRecord[LastSSPos-2].line/LinesPerMsec/WindowTime;
    currentframe := trunc(pastframes)+1;
    lineno := round((currentframe-pastframes)*WindowTime*LinesPerMsec);

    reset (Peak_frames);
    seek(Peak_frames, currentframe);

    While (lineno < LINESPERPAGE) do
    begin
      If not eof (Peak_frames) then
      begin
        read (Peak_frames,peak);
        For i := 1 to PEAKSPERFRAME do
        begin
          moveto (XORIGIN+round(peak[i].freq*PointsPerkHz), YORIGIN+lineno);
          linerel (0,0);
        End{For}
      End{If};
    end
  end

```

```

        inc(currentframe);
        lineno := round((currentframe-pastframes)*WindowTime*LinesPerMsec);
    End{While};
    End{Then}
    else repeat
        ch := prompt (400,50,'Error opening the Peak_frame file. '+
            'Press ESC',' ',1, escaped)
        until (ch[1] in [#27,#2]);
    End{Procedure DrawPeakPage};

```

```

Procedure DrawKey;
var
    i: classtype;
    x,y,j,thisx: integer;
    ch: char;
    textinfo: textsettingstype;
begin
    x := 500;
    y := 0;
    GetGraphInput (x,y,ch,[#1,#2,#27,#13],10, TRUE,0,0,getmaxx,getmaxy);
    If ch in [#1,#13] then
    begin
        gettextsettings(textinfo);
        settxtstyle(DefaultFont,VertDir,1);
        settxtjustify (TopText, RightText);

        for i := silent to mixed do
        begin
            thisx := x+10*ord(i);
            moveto(thisx+8,y+10);
            setcolor(white);
            OutInvertText(classname[i]);
            setcolor (ClassColour(i));
            for j := 0 to 7 do
            begin
                moveto(thisx+j,y);
                linerel(0,8);
            end;
            setcolor(white);
            rectangle (thisx,y,thisx+8,y+8);
        end;
        with textinfo do
        begin
            settxtstyle(Font,Direction, CharSize);
            settxtjustify(Horiz,Vert);
        end;
    End {If};
    End {DrawKey};

```

```

TYPE
    classplottype = (shapefreq,magnitude,frequency,freqxmag);
CONST
    shapefreqstr: string[11] = 'Reconstruct';
    frequencystr: string[9] = 'Frequency';
    magnitudestr: string[9] = 'Magnitude';
    freqxmagstr: string[13] = 'Product F x M';
    colourstr: string[10] = 'Colour ON';
    colour: boolean = TRUE;
    differencestr: string[15] = 'Diffrential OFF';
    Difference: boolean = FALSE;
    geometricstr: string[11] = 'Operation -';
    Geometric: boolean = FALSE;

```

```

VAR
    ClassPlot: classplottype;

```

```

Procedure SetClassShapeFreq;
begin
    ClassPlot := shapefreq;
    MenuQuitFlag := TRUE;
End;

```

```

Procedure SetClassMagnitude;
begin
    ClassPlot := magnitude;
    MenuQuitFlag := TRUE;
End;

```

```

Procedure SetClassFrequency;
begin
    ClassPlot := frequency;
    MenuQuitFlag := TRUE;
End;

```

```

Procedure SetClassFreqxMag;
begin
  ClassPlot := freqxmag;
  MenuQuitFlag := TRUE;
End;

Procedure SetClassColour;
begin
  Colour := not Colour;
  if Colour
  then colourstr := 'Colour ON'
  else colourstr := 'Colour OFF'
End;

Procedure SetClassDifferential;
begin
  Difference := not Difference;
  if Difference
  then differencestr := 'Differential ON'
  else begin
    differencestr := 'Differential OFF';
    Geometric := FALSE;
    geometricstr := 'Operation -';
  end;
End;

Procedure SetClassGeometric;
begin
  if Difference then {Geometric comparisons are only valid for differential}
    {plotting - for non-difference b will be 0}
  begin
    Geometric := not Geometric;
    if Geometric
    then geometricstr := 'Operation +'
    else geometricstr := 'Operation -';
  end;
End;

CONST
  GeneralScale: real = 1;
  scalestr: string[15] = 'Scale 1';
Procedure SetGeneralScale;
var
  f: real;
  fs: string[6];
  escaped: boolean;
  err: integer;
begin
  repeat
    fs := Prompt (500,140,'Enter plot scale:',
                  Strung(0,0,GeneralScale),6,escaped);
    val (fs,f,err);
  until (err=0) or escaped;

  if not escaped
  then begin
    GeneralScale := f;
    scalestr := 'Scale '+fs;
  end;
End;

Procedure DrawClassPage;
var
  currentframe: longint;
  lineno: word;
  oldclassframe, classframe: classifiedframe;
  pastframes: real;
  i: 1..PeaksPerFrame;
  ch: string[1];
  escaped: boolean;
  HarmSpacing: word;
  magscale: real;

{}Function Comp(a,b,scale:real): longint;
{}
{} () If Geometric is false, the difference between the bigger and the smaller
{} () a and b is returned, otherwise the ratio of the bigger over the smaller
{} () is returned. The result is multiplied by scale.
{}
{}var
{} () tempresult: real;
{}begin
{} () if not Geometric
{} () then tempresult := abs(a-b)
{} () else if a > b then tempresult := a/(b+1e-6) else tempresult := b/(a+1e-6);

```

```

({)if tempresult*scale > 32768*65535 {sqrt(maxlongint) - prevents error 207 on}
({)                                     {the product calculation}
({)then Comp := 32768*65535
({)else Comp := round(tempresult*scale);
({)End;

begin
  if classesexist
  Then begin
    if not OverPlot then cleardevice;
    pastframes := SSPosRecord[LastSSPos-2].line/LinesPerMsec/WindowTime;
    currentframe := trunc(pastframes)+1;
    lineno := round((currentframe-pastframes)*WindowTime*LinesPerMsec);
    magscale := (getmaxx-XORIGIN)/MaxMag;
    reset(Classified_frames);
    if currentframe > 0
    then seek (Classified_frames, currentframe-1);
    read(Classified_frames, oldclassframe);
    if currentframe = 0 then reset (Classified_frames);

Menu (600,100,8,
      @shapefreqstr, SetClassShapeFreq,
      @magnitudestr, SetClassMagnitude,
      @frequencyst, SetClassFrequency,
      @freqxmagstr, SetClassFreqxMag,
      @colourstr, SetClassColour,
      @differencestr,SetClassDifferential,
      @geometricstr, SetClassGeometric,
      @scalestr, SetGeneralScale,
      nil,null,nil,null);

if not Difference then
begin
  oldclassframe.freq := 0;
  oldclassframe.mag := 0;
  {if Difference is not on, there will be no history, ie oldclassframe
  must be disabled.}
end;

While (lineno < LINESPERPAGE) do
begin
  If not eof (Classified_frames) then
  begin
    read (Classified_frames,classframe);
    HarmSpacing := round(classframe.freq*PointsPerKHz);
    if Colour then setcolor(ClassColour(classframe.class));
    moveto(XORIGIN,YORIGIN+lineno);
    Case ClassPlot of
      shapefreq: begin
        moverel (HarmSpacing,0);
        Case classframe.class of
          silent::
            single:begin
              moverel (-1,0);
              linerel (3,0);
            end;
          harmo: begin
            for i := 1 to PeaksPerFrame do
            begin
              moverel (-3,0);
              linerel (1,0); moverel (2,0);
              linerel (1,0); moverel (2,0);
              linerel (1,0);
              moverel (Harmspacing,0);
            end;
          end;
          multi: begin
            moverel (-4,0);
            linerel (9,0);
          end;
          noise: begin
            moverel (-50,0);
            linerel (49,0);
            moverel (2,0);
            linerel (49,0);
          end;
        End;{Case}
      end;
    ($R-)
      magnitude: begin
        moverel (0,1);
        linerel (Comp(classframe.mag,oldclassframe.mag,
          GeneralScale*magscale),0);
      end;
  end;
end;

```

```

        frequency: linerel(Comp(classframe.freq,oldclassframe.freq,
                                GeneralScale*PointsPerKHz),0);
        freqxmag: linerel(Comp(classframe.freq,oldclassframe.freq,GeneralScale)*
                           Comp(classframe.mag,oldclassframe.mag,Generalscale),0);
{$R+}
        End{Case};
    End{If};

    inc (currentframe);
    lineno := round((currentframe-pastframes)*WindowTime*LinesPerMsec);
    if Difference then oldclassframe := classframe;
    End{While};
    setcolor(white);
    End{Then}

    else repeat
        ch := prompt (400,50,'Error opening the Classified_frame file.'+
                        ' Press ESC','','1, escaped)
        until (ch[1] in [#27,#2]);
    End{Procedure DrawPeakPage};

Procedure DrawElementPage;
var
    pastduration, currentduration: real;
    newlineno,lineno: word;
    element: elementtype;
    ch: string[1];
    escaped, fileend: boolean;
    i: classtype;
    blockstart: word;

const
    mainclasswidth = 20;
    totalclasswidth = 100;

{}Procedure Drawblock (width: word; clss: classtype);
{}
{} Draws a rectangle, vertically from 'lineno' to 'newlineno',
{} horizontally from 'blockstart', 'width' pixels in the colour belonging to
{} class 'clss' as defined by ClassColour.
{} All relative to (XORIGIN, YORIGIN). 'blockstart' then points to the
{} the rightmost edge of the block.
{}
{}var
{} rect: array [1..4] of record x,y:word end;
{} coloursave: word;
{}
{}begin
{} coloursave := getcolor;
{} with rect[1] do begin x := XORIGIN+blockstart; y := YORIGIN+lineno; end;
{} with rect[2] do begin x := rect[1].x+width; y := rect[1].y; end;
{} with rect[3] do begin x := rect[2].x; y := YORIGIN+newlineno; end;
{} with rect[4] do begin x := rect[1].x; y := rect[3].y; end;
{} inc(blockstart,width);
{} setcolor(ClassColour(clss));
{} setfillstyle(solidfill,ClassColour(clss));
{} fillpoly (4,rect);
{} setcolor(coloursave);
{}End; {}Procedure

begin
    if elementsexist
    then begin
        if not OverPlot then cleardevice;
        pastduration := SSPosRecord[LastSSPos-2].line/LinesPerMsec;
        lineno := 0;
        reset(Elements);

        currentduration := 0;
        While not eof(Elements) and (currentduration <= pastduration) do
        begin
            read(Elements,element);
            currentduration := currentduration+element.durn;
        End;

        Repeat
            fileend := eof(Elements);
            newlineno := round((currentduration-pastduration)*LinesPerMsec);

            blockstart := 0;
            Drawblock (mainclasswidth, element.class);
            blockstart := blockstart+5;
            For i := single to noise do
                Drawblock (round(totalclasswidth*element.occurrence[i]),i);

```

```

        if not fileend then read (Elements,element);
        currentduration := currentduration + element.durn;
        lineno := newlineno;
        Until (lineno >= LINESPERPAGE) or fileend;
    end {then}
else repeat
    ch := prompt (400,50,'Error opening the Elements file.'+
        ' Press ESC',' ',1, escaped)
    until (ch[1] in [#27,#2]);
End {Procedure DrawElementPage};

TYPE
    classarray = array [1..$FFFF] of classtype; {Will be used with 'classes'}

CONST
    MAXSEGMENTLENGTH = 100;
    ClassAnalysed: classtype = silent; {the class currently being searched for}
    PlotNo: byte = 1; {the number of plots since the last screen
        clear -determines the colour of the plot}
    StartSegLength: 2..MAXSEGMENTLENGTH = 2; {the first segmnt length to analyse}
    EndSegLength: 2..MAXSEGMENTLENGTH = 2; {the last segmnt length to analyse}
    AnalysisStep: 1..MAXSEGMENTLENGTH = 1; {segment length step}
    ProportionalMode: boolean = FALSE; {determines whether to plot the sum of}
        {classes in the segment as a}
        {proportion of the segment length}
        {Stores the classes in the classified}
        {frames file}

    classes: ^classarray = NIL;

    classtr: string[12] = 'Class Silent';
    fromstr: string[15] = 'From 2 frames';
    tostr: string[13] = 'To 2 frames';
    stepstr: string[15] = 'Jump 1 frames';
    modestr: string[12] = 'Mode Normal';
    plotstr: string[9] = 'Analysis Plot';

Procedure PlotElementAnalysis;
{
    Plots the analysis of the class groupings based on counting the number of
    class 'ClassAnalysed' frames occurring within a segment of 'FramesAnalysed'
    frames, plotted against the time at which the segment starts.
}

const
    DISPLACESCALE = 10;

var
    currentframe: longint;
    lineno: word;
    pastframes: real;

    i:word;
    SegSum:0..MAXSEGMENTLENGTH;
    StartSeg, EndSeg: 1..$FFFF;
    SegLength: 2..MAXSEGMENTLENGTH;
    colorsave: byte;
    viewinfosave: viewporttype;

begin
    if not OverPlot then cleardevice;
    getviewsettings(viewinfosave);
    colorsave := getcolor;

    SegLength := StartSegLength;
    setcolor (ord(ClassAnalysed)*3+1);

    Repeat
        {locate the current frame number based on the page currently viewed}
        pastframes := SSPosRecord[LastSSPos-2].line/LinesPerMsec/WindowTime;
        currentframe := trunc(pastframes)+1;

        {now set up the segment and put the current frame in the middle}
        StartSeg := currentframe;
        EndSeg := StartSeg + SegLength-1;
        inc(currentframe,SegLength div 2);

        {Note - the current frame is placed at the anchor point of the segment
        which was arbitrarily defined here as its centre. This affects the t-axis
        displacement of the various pekaks and valleys in the plot}

        {Calculate the initial segment sum}
        SegSum := 0;
        For i := StartSeg to EndSeg do
            if classes^[i] = ClassAnalysed then inc(SegSum);

```

```

{Locate the screen line corresponding to the current frame}
lineno := round((currentframe-pastframes)*WindowTime*LinesPerMsec);

setviewport (XORIGIN+round((currentframe-pastframes))*DISPLACESCALE,YORIGIN,
            getmaxx,getmaxy,ClipOff);
{This is to get a quicker plot by not having to do the displacement calcs.}

{The loop for one contour plot}
While (lineno < LINESPERPAGE) and
      (EndSeg < filesize(Classified_frames)) do
begin
  {uncount the outgoing class frame}
  if classes^[StartSeg] = ClassAnalysed then dec(segsum);
  inc(StartSeg);
  inc(EndSeg);
  if classes^[EndSeg] = ClassAnalysed then inc(segsum);

  if ProportionalMode
  then lineno := round(segsum/SegLength*GeneralScale),lineno)
  else lineno := round(segsum*GeneralScale),lineno);

  inc(currentframe);
  lineno := round((currentframe-pastframes)*WindowTime*LinesPerMsec);
End{While};

inc(SegLength,AnalysisStep);
Until (SegLength > EndSegLength);

with viewinfosave do setviewport (x1,y1,x2,y2,clip);
setcolor (colorsave);
End{Procedure PlotElementAnalysis};

Procedure PageBackHalf;
begin
  if LastSSPos > 2 then dec(LastSSPos,3) else LastSSPos := 0;
  totallines := SSPosRecord[LastSSPos].line;
  Seek (Sonagram_storage, SSPosRecord[LastSSPos].pos);
  DrawSonaPage;
End{Procedure PageBackHalf};

Procedure RedrawPage;
begin
  dec(LastSSPos,2);
  totallines := SSPosRecord[LastSSPos].line;
  Seek (Sonagram_storage, SSPosRecord[LastSSPos].pos);
  DrawSonaPage;
End{Procedure RedrawPage};

Procedure ChooseInt (Edge:intervaledge);
var
  px,py: integer;
  inkey: char;

begin
  px := XORIGIN; py := YORIGIN;
  if Edge = Endint then inc(py,LINESPERPAGE);
  GetGraphInput(px, py, inkey,
               [#1,#2,#27, #13],10,true,
               XORIGIN, YORIGIN, 600, YORIGIN+LINESPERPAGE);

  If inkey in [#13,#1] then
  begin
    Interval[Edge].line := SSPosRecord[LastSSPos-2].line+py-YORIGIN;
    RedrawPage;
  End{If};
End{Procedure ChooseInt};

Procedure ChooseBegInt;
begin
  ChooseInt (Begint);
End{Procedure};

Procedure ChooseEndInt;
begin
  ChooseInt (Endint);
End{Procedure ChooseEndInt};

Procedure FindBackGround;
begin
  Background(GlobalCallName, GlobalSpecName,
            1+round(Interval[Begint].line/LinesPerMsec/WindowTime),
            1+round(Interval[Endint].line/LinesPerMsec/WindowTime));
  MenuQuitFlag := TRUE; {break out of menu}
End{Procedure FindBackGround};

```

```

Procedure IntervalMenu;
const
  beginningstr: string[10] = 'Beginning';
  endstr: string[3] = 'End';
  findbackstr: string[12] = 'Find Backgnd';
begin
  Menu (500,37,3,
    @beginningstr, ChooseBegInt,
    @endstr, ChooseEndInt,
    @findbackstr, FindBackGround,
    nil,null,
    nil,null,
    nil,null,
    nil,null,
    nil,null,
    nil,null,
    nil,null);
End(Procedure IntervalMenu);

Procedure SetAnalysisClass;
begin
  if ClassAnalysed < noise then ClassAnalysed := succ(ClassAnalysed)
    else ClassAnalysed := classtype(0);
  classstr := 'Class '+classname[ClassAnalysed];
End;

Procedure SetStartLength;
var
  f:byte;
  fs: string[3];
  escaped: boolean;
  err: integer;
begin
  repeat
    fs := Prompt (100,100,'Number of frames for first analysis?',
      Strung(0,0,StartSegLength),3,escaped);
    val (fs,f,err);
  until ((err=0) and ((f>=2) and (f<=MAXSEGMENTLENGTH))) or escaped;

  if not escaped
  then begin
    StartSegLength := f;
    fromstr := 'From '+fs+ ' frames';
  end;
End;

Procedure SetEndLength;
var
  f:byte;
  fs: string[3];
  escaped: boolean;
  err: integer;
begin
  repeat
    fs := Prompt (100,100,'Number of frames for last analysis?',
      Strung(0,0,StartSegLength),3,escaped);
    val (fs,f,err);
  until ((err=0) and ((f>=2) and (f<=MAXSEGMENTLENGTH))) or escaped;

  if not escaped
  then begin
    EndSegLength := f;
    tostr := 'To '+fs+ ' frames';
  end;
End;

Procedure SetAnalysisMode;
begin
  ProportionalMode := not ProportionalMode;
  if proportionalmode then modestr := 'Mode Proportion'
    else modestr := 'Mode Normal';
End;

Procedure SetAnalysisStep;
var
  f:byte;
  fs: string[3];
  escaped: boolean;
  err: integer;
begin
  repeat
    fs := Prompt (100,100,'Segment length step?',

```

```

        Strung(0,0,AnalysisStep),3,escaped);
    val (fs,f,err);
until ((err=0) and (f >= 1) and
      (StartSegLength+f*(EndSegLength-StartSegLength) <= MAXSEGMENTLENGTH))
    or escaped;

if not escaped
then begin
    AnalysisStep := f;
    stepstr := 'Jump '+fs+ ' frames';
end;
End;

Procedure ElementAnalysisMenu;
var
    i: word;
    c: classifiedframe;
    ch: string[1];
    escaped: boolean;
begin
    {First read the classes (not the freqs) in the classified frames file into
    a heap array, so that access is quicker for the plots that follow. Note
    that the filesize is assumed to be <= $FFFF, even though the currentframe
    variable that usually counts frames has always been a longint. In practice,
    64000 frames of 1 ms is 64 seconds, so there is no problem.
    filesize(Classified frames) could be replaced by NumberOfFrames, a
    global which could be read from the BCD Number)of_frames - but what for}

    if classesexist
    then begin
        reset (Classified frames);
        getmem (classes, filesize(Classified frames));
        for i := 1 to filesize(Classified_frames) do
        begin
            read (Classified_frames,c);
            classes^[i] := c.class;
        end;

        Menu(500,37,7,
            @classstr,    SetAnalysisClass,
            @modestr,     SetAnalysisMode,
            @fromstr,     SetStartLength,
            @tostr,       SetEndLength,
            @stepstr,     SetAnalysisStep,
            @scalestr,    SetGeneralScale,
            @plotstr,     PlotElementAnalysis,
            nil,null,
            nil,null,
            nil,null);

        freemem (classes, filesize(Classified_frames));
        close(Classified_frames);
    end
    else repeat
        ch := prompt (400,50,'Error opening the Classified_frame file.'+
            ' Press ESC',' ',1, escaped)
        until (ch[1] in [#27,#2]);
    End;{Procedure ElementAnalysisMenu}

Procedure ToggleOverPlot;
begin
    OverPlot := not OverPlot;
    if OverPlot then overplotstr := 'Overplot ON'
    else overplotstr := 'Overplot OFF'
End;

Procedure FunctionMenu;
const
    intervalstr: string[8] = 'Interval';
    screendumpstr: string[12] = 'Dump screen';
    graphaxesstr: string[10] = 'Graph axes';
    classkeystr: string[9] = 'Class key';
begin
    Menu(500,60,4,
        @intervalstr, IntervalMenu,
        @screendumpstr, ScreenDump,
        @graphaxesstr, AxisMenu,
        @classkeystr, DrawKey,
        nil,null,nil,null,nil,null,nil,null,nil,null,nil,null);
End;

{$F-}

```

```

Procedure Display;
var
  mode: integer;
  WindowLength: word;
  SamplingFreq: real;
  ch: string[1];
  escaped: boolean;

const
  nextpagestr:string[10] = 'Next Page';
  backhalfstr:string[12] = 'Back '+#171;

  sonagramstr: string[9] = 'Sonagram';
  peaksstr: string[10] = 'Peaks';
  classstr: string[12] = 'Classes';
  elementstr: string[12] = 'Elements';
  analysisstr: string[12] = 'Analysis';
  functionstr: string [12] = 'Functions';

begin
  PutCurrentProcess('DRAWER');

  assign (Sonagram_Storage, CallName+SonaName+'.SS');
  if FileQuery (Sonagram_Storage,DontAsk) = Exists
    then reset (Sonagram_Storage)
    else GenRunError (0,'DRAWER could''nt find the '+CallName+SonaName+'.SS file');

  BCD('read '+CallName+'.ADS '+'Sampling Frequency: real', SamplingFreq);
  BCD('read '+CallName+SpecName+'.SF '+'Window_length: word', WindowLength);
  BCD('read '+CallName+SpecName+'.SF '+'CallName+SonaName+'.SS '+'PixelsPerkHz: real',
    PointsPerkHz);
  BCD('read '+CallName+SpecName+'.SF '+'CallName+SonaName+'.SS '+'LinesPerMsec: real',
    LinesPerMsec);
  BCD('read '+CallName+SpecName+'.SF '+'Max_magnitude: real', MaxMag);

  WindowTime := WindowLength/SamplingFreq;

  assign (Peak_frames, CallName+PeakName+'.LPF');
  peaksexist := (FileQuery(Peak_frames, DontAsk) = Exists);

  assign (Classified_frames,CallName+ClassName+'.CF');
  classesexist := (FileQuery(Classified_frames, DontAsk) = Exists);

  assign (Elements,Callname+ElementName+'.E');
  elementsexist := (FileQuery(Elements,DontAsk) = Exists);

  GlobalCallName := CallName;
  GlobalSpecName := SpecName;

  detectgraph (driver,mode);

  {$ifdef EGALO}
  if (driver in [Ega,Ega64]) and (mode in [EgaLo, Ega64Lo]) then mode := egalo;
  {$endif}

  initgraph (driver,mode,'');
  LINESPERPAGE := getmaxy -{bottom margin}20 -{top margin}YORIGIN;
  {leave space at top for axis labels and bottom for extra axis which
   helps join the graphs}

  if (driver = ega) then
  begin
    setpalette(0,0);
    setpalette(1,4);
    setpalette(2,2);
    setpalette(3,6);
    setpalette(4,1);
    setpalette(5,5);
    setpalette(6,3);
    setpalette(7,7);
    setpalette(8,0+16);
    setpalette(9,4+16);
    setpalette(10,2+16);
    setpalette(11,6+16);
    setpalette(12,1+16);
    setpalette(13,5+16);
    setpalette(14,3+16);
    setpalette(15,7+16);
  end;

```

```

totallines := 0;
LastSSPos := 0;
fillchar (SSPosRecord,sizeof(SSPosRecord),#0);
SSPosRecord[0].line := 0;
SSPosRecord[0].pos := filepos(Sonagram_storage);

DrawSonaPage;

Repeat
  Menu (539,0,9,
    @nextpagestr, DrawSonaPage,
    @backhalfstr, PageBackHalf,
    @overplotstr, ToggleOverPlot,
    @sonagramstr, RedrawPage,
    @peaksstr, DrawPeakPage,
    @classstr, DrawClassPage,
    @elementstr, DrawElementPage,
    @analysisstr, ElementAnalysisMenu,
    @functionstr, FunctionMenu,
    nil,null);

  Until Upstring(Prompt (600,100,'Exit or resume program? (E/anykey)','',1,escaped))
    = 'E';

close (Sonagram_storage);
closegraph;
End{Procedure Display};

BEGIN{INITIALIZATION}
  xlbox := 0;   ylbox := 0;
  x2box := 100; y2box := 100;
  fAxis := YORIGIN; tAxis := XORIGIN;
  EndfAxis := XORIGIN; EndtAxis := YORIGIN;
  Graphtitle := '';
  FileName := '';
END{UNIT}.

```

B1.14 SOUNDGEN.PAS

```
UNIT SOUNDGEN;

INTERFACE
USES BCRTYPES;

Procedure GenerateSounds(CallName, SpecName, PeakName, DASName: NameStr);

{
  Executes SOUNDGEN process.

  Generates the time waveform corresponding to the input Linked Peak Frame
  (.LPF) file. Outputs a file of Digital-to-Analogue Samples (.DAS), identical
  in format to the Analogue-to-Digital (.ADS) files used as input to SAMPLE.
  The .DAS file is intended for playback or to be otherwise processed as
  a bird call.

  'CallName' is the base bird file name, 'SpecName' is the Spectral Frames (.SF)
  suffix for the file from which the input .LPF file was derived, 'DASName' is
  the suffix for the output .DAS file.
}

IMPLEMENTATION
USES Crt, Graph, BCDMANIP, GENLIB;

TYPE
  MaxWindowLength = 1..MAXWINDOW;
  TimeFrame = array [MaxWindowLength] of real;

Procedure GenerateSine (StartFreq, EndFreq: real; StartMag, EndMag: real;
  var PhaseAngle: real;
  TimeStep: real; Samples: MaxWindowLength;
  var Result: TimeFrame);
{
  Generates a sinusoidal wave segment with starting phase 'PhaseAngle'. The
  frequency and amplitude start at 'StartFreq' and 'StartMag' and change
  linearly with time, finally reaching 'EndFreq' and 'EndMag'. 'SampleNo'
  points are generated, each being 'TimeStep' seconds apart. Frequency is in
  Herz. The phase angle the last point plotted is returned in 'PhaseAngle', to
  allow continuity of phase between consecutive linked segments.
}
{
  Note peak values in milliseconds and kHz complement each other and can be
  used without conversion to seconds and Hz
}

var
  TwoPiTimeStep, Aslope, Fslope, Amplitude, Frequency: real;
  i: MaxWindowLength;
begin
  Fslope := (EndFreq-StartFreq)/Samples;
  Aslope := (EndMag-StartMag)/Samples;
  TwoPiTimeStep := 6.283185307*TimeStep;

  Amplitude := StartMag;
  Frequency := StartFreq;

  For i := 1 to Samples do
  begin
    Result[i] := Amplitude*sin(PhaseAngle);
    Amplitude := Amplitude+Aslope;
    Frequency := Frequency+Fslope;
    PhaseAngle := PhaseAngle + TwoPiTimeStep*Frequency;
  End{For};
End{Procedure GenerateSine};

Procedure GenerateSounds;
type
  peakandphase = record
    freq, mag, phase: real;
  end;
  QTimeFrame = array [MaxWindowLength] of integer;
var
  ComponentSeg, CompositeSeg: TimeFrame;
  QuantisedSeg: QTimeFrame;
  WindowLength: MaxWindowLength;
  SamplingFreq: real;
  MaxMag: real;
  frame, NoOfFrames: longint;
  PrevFrame: array [1..PeaksPerFrame] of peakandphase;
  CurrFrame: peakframe;
  Linked_peak_frames: file of peakframe;
```

```

DA_samples: file;
link: 0..PeaksPerFrame;
i,k: 1..PeaksPerFrame;
j: MaxWindowLength;
qfactor: real;

begin
  PutCurrentProcess ('SOUNDGEN');

  BCD ('read '+CallName+'.ADS Sampling_frequency: real', SamplingFreq);
  BCD ('read '+CallName+SpecName+'.SF Window_length: word', WindowLength);
  BCD ('read '+CallName+SpecName+'.SF Number_of_frames: longint', NoOfFrames);
  BCD ('read '+CallName+SpecName+'.SF Max Magnitude: real', MaxMag);
  assign (Linked_peak_frames, CallName+PeakName+'.LPF');
  assign (DA_samples, CallName+DASName+'.DAS');

  If FileQuery(DA_samples, AskOverWrite) in [DoesNotExist, OverWriteOK]
    then rewrite (DA_samples, WindowLength*sizeof(QuantisedSeg[1]))
    else GenRunError (0, 'SOUNDGEN could''nt find the DA_samples file');

  reset (Linked_peak_frames);

  frame := 1;
  gotoxy (1,8); writeln ('Soundgenning from frame          of ', NoOfFrames);
  fillchar (PrevFrame, sizeof (PrevFrame), #0);

  While not eof (Linked_peak_frames) do
  begin
    gotoxy (25,8); writeln (frame);
    read (Linked_peak_frames, Currframe);
    fillchar (CompositeSeg, sizeof (CompositeSeg), #0);

    For i := 1 to PeaksPerFrame do {generate a component segment}
    begin
      link := CurrFrame[i].link;
      if link <> 0
      then GenerateSine (PrevFrame[link].Freq, CurrFrame[i].Freq,
                        PrevFrame[link].Mag, CurrFrame[i].Mag,
                        PrevFrame[link].Phase, 1/SamplingFreq, WindowLength,
                        ComponentSeg)
      else begin
        PrevFrame[i].Phase := 0;
        GenerateSine (CurrFrame[i].Freq, CurrFrame[i].Freq,
                      CurrFrame[i].Mag, CurrFrame[i].Mag,
                      PrevFrame[i].Phase, 1/SamplingFreq, WindowLength,
                      ComponentSeg)
      end;

      {accumulate the composite segment for the entire frame}
      For j := 1 to WindowLength do
        CompositeSeg[j] := CompositeSeg[j] + ComponentSeg[j];
    End{For};

    For i := 1 to PeaksPerFrame do
    begin
      PrevFrame[i].Freq := CurrFrame[i].Freq;
      PrevFrame[i].Mag := CurrFrame[i].Mag;
    End{For};

    {quantise the range -MaxMag..MaxMag to the range -2048..2047, ie
     12 bits.}
    qfactor := 2047/MaxMag;
    For j := 1 to WindowLength do
      QuantisedSeg[j] := round(qfactor*CompositeSeg[j]);

    blockwrite (DA_samples, QuantisedSeg, 1);
    inc(frame);
  End{While};

  close (Linked_peak_frames);
  close (DA_Samples);
End{Procedure GenerateSounds};

END{UNIT}.

```

B1.15 LINKPEAK.PAS

```
PROGRAM LINKPEAKS;
{
  Executes LINKPEAK process.
  Utility for viewing and editing and printing a Linked Peak Frames (.LPF)
  file. Links between peaks may be added, removed or changed with this program.

  Usage: LINKPEAK [Linked Peak File name, no extension]

  Example Usage: LINKPEAK bird1a
  -Operate on BIRD1A.LPF, if it exists, else file to use will be prompted for.
  Example usage: LINKPEAK
  -File to operate on will be prompted for
}

USES Dos, Graph, BCRTYPES, BCDMANIP, PRTEGA, ICON, GENLIB;
CONST
  modifystr: string[6] = 'Modify';           {Main menu}
  clearlinksstr: string[11] = 'Clear links'; {Main menu}
  editstr: string[6] = 'Edit';               {Main menu}
  dumpscreenstr: string[11] = 'Dump screen'; {Main menu}
  leftedgeconst = 'L edge' + #179 + #27;
  rightedgeconst = 'R edge' + #26 + #179;
  inactiveconst = '.....';
  leftedgestr: string[9] = leftedgeconst;    {Modify menu} {These are global }
  rightedgestr: string[11] = inactiveconst;  {Modify menu} {because they are }
  expandcontractstr: string[8] = 'Expand';   {Modify menu} {changed by procs }
                                              {execeuted by Menu}

VAR
  Linked_peak_frames: file of peakframe;
  NoOfFrames: longint; {The value read in from BCD}
  Xscale, Yscale: byte; {The current plotting scale}
  StartFrame: longint; {The frame from which plotting begins}
  OldLeftEdge, OldRightEdge: integer; {Last edge pointed to}
  OldEditX, OldEditY: integer; {Last place edited}
  escaped: boolean; {Needed for Prompt call in main program}
  driver, mode: integer;
  Dir: DirStr;
  Name: Dos.NameStr;
  Ext: ExtStr;

Procedure DrawLinkedPeaks;
{
  Plots the data in Linked_peak_frames (assumes the file is open) from
  StartFrame until end of screen or file.
}
var
  i: 1..PeaksPerFrame;
  frame: longint;
  oldlpf, lpf: peakframe;
begin
  cleardevice;
  frame := 1;
  fillchar (oldlpf, sizeof(oldlpf), #0);
  seek (Linked_Peak_Frames, StartFrame);
  While (frame*xscale <= getmaxx) and not eof (Linked_peak_frames) do
    begin
      read(Linked_peak_frames, lpf);
      if frame = 1 then for i := 1 to PeaksPerFrame do lpf[i].link := 0;
      for i := 1 to PeaksPerFrame do
        begin
          moveto (round(frame*xscale), getmaxy-round(lpf[i].freq*yscale));
          if lpf[i].link <> 0
            then lineto (round((frame-1)*xscale),
                       getmaxy-round(oldlpf[lpf[i].link].freq*yscale))
            else linerel(0,0);
          end;
          inc 'frame';
          oldlpf := lpf;
        end
      End(While);
    end
  End(Procedure DrawLinkedPeaks);

Function Pointed (var x,y: integer; var RelFrame:longint): Boolean;
{
  Allow user to point to a frame in the displayed portion of the call.
  Takes as input the global variables Xscale, Yscale which it uses to calculate
  the frame no. relative to the start of the screen, returned in RelFrame.
  The pointing begins at the position given by (x,y), and the position
  on exit is returned in (x,y). If ENTER (or mouse LEFT) is pressed the current
  position is accepted. If ESC (right) is pushed no change is made to the
  output variables, but Pointed returns FALSE;
}
```

```

}
var
  returnkey: char;

begin
  GetGraphInput (x, y, returnkey, [#1,#2,#13,#27],
    10,TRUE,
    0,0,getmaxx, getmaxy);
  if returnkey in [#1,#13]
  then begin
    RelFrame := Round(x/Xscale);
    Pointed := TRUE;
  end
  else Pointed := FALSE;
End{Procedure Pointed};

{$F+} {The following procedures will be called via procedure variables}

{Stuff for screen dumping}
CONST
  FileName: string[MaxLineLength] = '';
  printer: printertype = PostScript;
  printerstr: string[12] = 'PostScript';
  printscale: printscaletype = Normal;
  printscalestr: string[12] = 'Scale Normal';
  printform: printformtype = Full;
  printformstr: string[12] = 'Format Full';

  x1box: integer = 0;
  x2box: integer = 200;
  y1box: integer = 200;
  y2box: integer = 300;

Procedure TogglePrinter;
begin
  Case printer of
    LaserJet2: begin
      printer := Epson;
      printerstr := 'Epson compat';
    end;
    Postscript: begin
      printer := LaserJet2;
      printerstr := 'HP LaserJet';
    end;
    Epson: begin
      printer := Postscript;
      printerstr := 'PostScript';
    end;
  End{Case}
End{Procedure TogglePrinter};

Procedure TogglePrintScale;
begin
  Case printscale of
    Stretched: begin
      printscale := Normal;
      printscalestr := 'Scale Normal';
    end;
    Normal: begin
      printscale := Squashed;
      printscalestr := 'Scale Squash';
    end;
    Squashed: begin
      printscale := Stretched;
      printscalestr := 'Scale Strtch';
    end;
  End{Case}
End{Procedure TogglePrintScale};

Procedure TogglePrintForm;
begin
  Case printform of
    Full: begin
      printform := Split;
      printformstr := 'Format Split';
    end;
    Split: begin
      printform := Full;
      printformstr := 'Format Full';
    end;
  End{Case}
End{Procedure TogglePrintForm};

```

```

Procedure DumpToFile;
var
    selected,decentname, escaped:boolean;
    errmess:string[60];
    tempfile: file of byte;
begin
    ChooseBox (xlbox,ylbox,x2box,y2box,selected);
    If selected then
        begin
            errmess := '';
            Repeat
                FileName := Prompt (500,40,errmess+'Destination file name:',
                                    FileName,MaxLineLength, escaped);

                If not escaped then
                    begin
                        { $I- }
                        assign (tempfile,FileName);
                        rewrite (tempfile);
                        close (tempfile);
                        { $I+ }
                        if IOresult <> 0
                        Then begin
                            errmess := 'File creation error: redo ... ';
                            decentname := FALSE;
                        End{Then}
                        Else decentname := TRUE;
                    End{If};
                Until escaped or decentname or (FileName = '');

                if not escaped and (FileName <> '')
                then PrintEga (FileName,printer,printscaled, printform,
                             xlbox,ylbox,x2box,y2box);

                MenuQuitFlag := TRUE;
            End{If};
        End{Procedure DumpToFile};

Procedure ScreenDump;
const
    dumpstr: string[10] = 'Begin Dump';
begin
    Menu (500,57,4,
          @dumpstr,      DumpToFile,
          @printerstr,   TogglePrinter,
          @printscaledstr, TogglePrintScale,
          @printformstr, TogglePrintForm,
          nil,null,
          nil,null,
          nil,null,
          nil,null,
          nil,null,
          nil,null);
End{Procedure ScreenDump};
{End of stuff for screen dumping}

Procedure ExpCon;
{
    change xscale and yscale to effect an expanded view, then call
    DrawLinkedPeaks to show changes.
}
begin
    if expandcontractstr = 'Expand'
    then begin
        Xscale := 5;
        Yscale := getmaxy div 8; {8 kHz is the highest likely frequency}
        expandcontractstr := 'Contract';
    end
    else begin
        Xscale := 2;
        Yscale := getmaxy div 16;
        expandcontractstr := 'Expand';
    end;

    if StartFrame >= NoOfFrames - getmaxx div Xscale
    then leftedgestr := inactiveconst
    else leftedgestr := leftedgeconst;
    if StartFrame <= 0
    then rightedgestr := inactiveconst
    else rightedgestr := rightedgeconst;
    DrawLinkedPeaks;
End{Procedure ExpCon};

```

```

Procedure LeftEdge;
var
  y: integer;
  relframe, lastleftedge: longint;
begin
  y := getmaxy div 2;
  lastleftedge := NoOfFrames-getmaxx div xscale;
  if StartFrame < lastleftedge then
    If Pointed (OldLeftEdge, y, relframe) then
      begin
        inc(StartFrame, relframe);
        If StartFrame >= lastleftedge then
          begin
            StartFrame := lastleftedge;
            leftedgestr := inactiveconst;
          End{If};
          if StartFrame > 0 then rightedgestr := rightedgeconst;
          DrawLinkedPeaks;
        End{If};
      End{Procedure LeftEdge};

Procedure RightEdge;
var
  y: integer;
  relframe, lastleftedge: longint;
begin
  y := getmaxy div 2;
  lastleftedge := NoOfFrames-getmaxx div xscale;
  if StartFrame > 0 then
    If Pointed (OldRightEdge, y, relframe) then
      begin
        dec(StartFrame, round(getmaxx/xscale)-relframe);
        If StartFrame <= 0 then
          begin
            StartFrame := 0;
            rightedgestr := inactiveconst;
          End{If};
          if StartFrame < lastleftedge then leftedgestr := leftedgeconst;
          DrawLinkedPeaks;
        End{If};
      End{Procedure RightEdge};

Procedure EditFrames;
{
  Open an edit window containing the first 'editframeno' frames after the
  pointed-to frame, then allow user to choose a peak in any of the frames
  and link it to one (or none) in the previous frame.
}
const
  editframeno = 30;
type
  OneToEditFrameNo = 1..editframeno;
  ZeroToPeaksPerFrame = 0..PeaksPerFrame;
  OneToPeaksPerFrame = 1..PeaksPerFrame;
var
  f, edfn, curfrm: 1..editframeno;
  i, curpk: 1..PeaksPerFrame;
  ef: array[1..editframeno] of peakframe;
  editxscale, edityscale: word;
  x, y, edwinheight, edwinwidth: integer;
  returnkey: char;
  endedit: boolean;
  editwindow: viewporttype;
  oldlink, newlink: ZeroToPeaksPerFrame;
  delaycount: word;
  esc: boolean;
  chstr: string[1];
  relframe: longint;

{}Procedure ShowEditFrames;
{}
{} Display the frames in the 'ef' variable in the edit window.
{}
{}var
{}  i: 1..PeaksPerFrame;
{}  f: 1..editframeno;
{}begin
{}  clearviewport;
{}  for f := 1 to edfn do
{}    For i := 1 to PeaksPerFrame do
{}      begin
{}        moveto(f*editxscale, edwinheight-round(ef[f,i].freq*edityscale));
{}        if (ef[f,i].link <> 0) and (f>1)
{}          then lineto ((f-1)*editxscale,
{}                      edwinheight-round(ef[f-1,ef[f,i].link].freq*edityscale))

```

```

() else linerel(0,0);
() End{For};
()End{Procedure ShowEditFrames};

()Function NearestPeak(f,y: integer): ZeroToPeaksPerFrame;
()
() Finds, in the f'th frame of 'ef', the nearest peak to the height indicated
() by y. Uses the 'edityscale' value. If y is in the "zero area" then 0 is
() returned as the peak number.
()
()var
() i: 1..PeaksPerFrame;
() tempdist, mindist: real;
() nearpeak: 1..PeaksPerFrame;
()begin
() mindist := 1e37;
() For i := 1 to PeaksPerFrame do
() begin
() tempdist := abs(ef[f,i].freq - (edwinheight-y)/edityscale);
() if tempdist < mindist then begin
() mindist := tempdist;
() nearpeak := i;
() end;
() End{For};
() NearestPeak := nearpeak;
()End;

()Procedure DrawLink (cf: OneToEditFrameNo; cp: OneToPeaksPerFrame;
() link: ZeroToPeaksPerFrame; colour: byte);
()
() Draws the link between the current peak ('cp') in the current frame ('cf')
() of 'ef' and the link'th peak in the previous frame. The line is drawn in
() colour 'colour' and displaced in the y direction by 'displacement'.
()
()begin
() If link <> 0 then
() begin
() setcolor (colour);
() line(cf*editxscale,
() round(edwinheight-ef[cf,cp].freq*edityscale), (cf-1)*editxscale,
() round(edwinheight-ef[cf-1,link].freq*edityscale));
() setcolor (15);
() End{If};
()End{Procedure DrawLink};

begin
editxscale := 5;
edityscale := round(getmaxy/8);

If Pointed (OldEditX, OldEditY, relframe) then
begin
seek (Linked_peak_frames,Startframe+relframe);
edfn := editframenno; (counts actual no of frames available - in case of eof)

{read in the frames to edit}
for f := 1 to editframenno do
if not eof(linked_peak_frames)
then read(Linked_peak_frames, ef[f])
else dec(edfn);

{Set an edit window viewport, draw window border}
setviewport (getmaxx-edfn*editxscale-11, 1,getmaxx-9, getmaxy-1, clipoff);
getviewsettings(editwindow);
With editwindow do
begin
edwinheight := y2-y1;
edwinwidth := x2-x1;
End{With};
rectangle (-1,-1,edwinwidth+1,edwinheight+1);

ShowEditFrames;
chstr := Prompt (getmaxx-editframenno*xscale,0,'Remove links? (Y/any)', '',
1,esc);
if chstr[1] in ['y','Y'] then
for f := 1 to edfn do
for i := 1 to PeaksPerFrame do ef[f,i].link := 0;

(edit loop)
x := edwinwidth div 2; y := edwinheight div 2;
endedit := FALSE;
Repeat
ShowEditFrames;
GetGraphInput (x,y,returnkey,[#1,#2,#13,#27],10,TRUE,
0,0,edwinwidth,edwinheight);
if returnkey in [#2,#27]

```

```

then endedit := TRUE
else Begin
    if round(x/editxscale) < 2
    then curfrm := 2
    else curfrm := round(x/editxscale);
    curpk := NearestPeak(curfrm,y);
    oldlink := ef[curfrm,curpk].link;
    if oldlink = 0 then oldlink := NearestPeak(curfrm-1,y);

    {Flash the chosen line (for Hercules - colour doesn't change)}
    DrawLink (curfrm, curpk, oldlink,0);
    for delaycount := 0 to 20000 do;
    DrawLink (curfrm, curpk, oldlink,5);
    for delaycount := 0 to 20000 do;
    DrawLink (curfrm, curpk, oldlink,0);
    for delaycount := 0 to 20000 do;
    DrawLink (curfrm, curpk, oldlink,5);

    Repeat
        GetGraphInput (x,y, returnkey, [#1,#2,#13,#27,#32],10,FALSE,
            x,0,x,edwinheight);
        newlink := NearestPeak(curfrm-1,y);
        DrawLink (curfrm, curpk, oldlink, 0);
        DrawLink (curfrm, curpk, newlink, 5);
        oldlink := newlink;
    Until returnkey in [#1,#2, #13, #27, #32];
    Case returnkey of
        #32:ef[curfrm,curpk].link := 0;
        #1,#13:ef[curfrm,curpk].link := newlink;
    End{Case};
    End{Else};
Until endedit;

setviewport (0,0, getmaxx, getmaxy,clipoff);
seek (Linked_peak_frames,Startframe+relframe);
for f := 1 to edfn do write(Linked_peak_frames, ef[f]);
DrawLinkedPeaks;
End{If};
End{Procedure EditFrame};

Procedure ClearLinks;
var
    newlpf: file of peakframe;
    lpf: peakframe;
    i: 1..PeaksPerFrame;
begin
    assign (newlpf, 'NLPF.DAT');
    rewrite (newlpf);
    reset (Linked_peak_frames);
    While not eof(Linked_peak_frames) do
    begin
        read (Linked_peak_frames ,lpf);
        for i := 1 to PeaksPerFrame do lpf[i].link := 0;
        write (newlpf,lpf);
    End{While};
    close (newlpf);
    erase (Linked_peak_frames);
    assign (Linked_peak_frames, 'NLPF.DAT');
    rename (Linked_peak_frames, Name+'.LPF');
    reset (Linked_peak_frames);
    DrawLinkedPeaks;
End {Procedure ClearLinks};

{$F-}

BEGIN
    PutCurrentProcess ('LINKPEAK');

    driver := detect;
    initgraph (driver, mode, '');

    {Initialisation of global variables used by procedures}

    BCD('read Number_of_frames: longint',NoOfFrames);
    Xscale := 2;
    Yscale := getmaxy div 16;
    StartFrame := 0;
    OldEditX := getmaxx div 2;
    OldEditY := getmaxy div 2;

    if NoOfFrames <= getmaxx div Xscale then leftedgestr := inactiveconst;

```

```

escaped := FALSE;
fsplit (paramstr(1), Dir,Name,Ext);
If Name = '' then Name := 'noname';
assign(Linked_peak_frames,Name+'.LPF');
if FileQuery (Linked_peak_frames, DontAsk) <> Exists
then repeat
    fsplit (Prompt(0,40,'Linked peak file name? (no extension)',
        Dir+Name+Ext,8,escaped),Dir,Name,Ext);
    assign(Linked_peak_frames,Name+'.LPF');
until escaped or (FileQuery (Linked_peak_frames, DontAsk) = Exists);

if not escaped
then begin
    reset (Linked_peak_frames);
    DrawLinkedPeaks;
    Menu (0,0,6,
        @editstr, EditFrames,
        @expandcontractstr, ExpCon,
        @clearlinksstr, ClearLinks,
        @rightedgestr, RightEdge,
        @leftedgestr, LeftEdge,
        @dumpscreensstr, ScreenDump,
        nil, null,
        nil, null,
        nil, null,
        nil, null);
    close (Linked_peak_frames);
end;
closegraph;
END.

```

B1.16 BCDMANIP.PAS

UNIT BCDMANIP; {Bird call descriptor maintenance library}

INTERFACE

Procedure BCD(command:string; var value(:anytype));

{
Interface procedure to the Bird Call Descriptor, which must be stored in a file called BCD.DAT in the current directory. All access to BCD.DAT is through this procedure.

'command' contains an access command ("read" or "write"), a path of names (separated by spaces) and a type name. The last name in the path (second last in the 'command' string) is the field name, and should be colon-terminated. Each name in the path is scanned for from the position in BCD.DAT at which the previous path name was found. Once the field name has been located, it's value may be accessed (read or written) according to the access command, providing that access is allowed for the current process. The value is accessed as one of the Pascal simple types (according to the type name (the last word in 'command')). The value is written from or returned in the parameter 'value', which may be of any type, but which should in fact be of the type specified in 'command'. The current process name is assumed to be present in the file CRP.DAT in the current directory. The access permissions for each process are specified in the file BCDACCES.MAP in the current directory.

Example usage:

```
BCD ('read Seoul-Olympics Athletics 100m Gold: string', winner);  
if winner = 'Johnson' then winner := 'Lewis';  
date = 1992;  
BCD ('write Barcelona-Olympics Year: integer', date);  
}
```

Procedure PutCurrentProcess (process:string);

{
Registration procedure for processes wishing to use the Bird Call Descriptor.

Makes the process with name 'process' the current process by writing the name to CRP.DAT, the Current_process file.
}

Procedure GetCurrentProcess (var process:string);

{
'Return the current process name (stored in CRP.DAT)'
}

IMPLEMENTATION

USES GENLIB;

Function UpString (instring:string):string;

{
[convert the input string 'instring' to uppercase]
}
var i:byte;
begin
for i := 1 to length(instring) do instring[i] := upcase (instring[i]);
UpString := instring;
End(Function);

Procedure BCD;

type seqrec = record
pos:byte;
str:string;
end;

{})Function GetWord (var stringseq:seqrec):string;

{
([Return the part of 'stringseq.str' that begins at 'stringseq.pos' and
(ends just before a space (or the end of the string, whichever comes
(first) having removed leading spaces. Update 'stringseq.pos' to point to
(the space after the last character returned. If the end of the string
(was reached, set 'stringseq.pos' to 1 (the beginning of stringseq.str).]
(
(The effect of this function is that words separated by spaces are re-
(trieved sequentially, like numbers from a text file, with retrieval
(wrapping round to the start of the string. The "last word read" con-
(dition can thus be detected when the '.pos' field of the input 'seqrec'
(type variable equals 1 AFTER a call to 'GetWord'.
(
{})var
{}) endpos,spaces:byte;
{}) tempword:string;

```

{}
{}begin
{} with stringseq do tempword := copy (str,pos,length(str)-pos+1);
{}
{} spaces := length (tempword);
{} while (tempword[1] = ' ') and (length(tempword) > 0) do
{}   tempword := copy(tempword,2,length(tempword)-1);
{}   dec(spaces,length(tempword));
{}
{} endpos := pos(' ',tempword);
{} if endpos = 0 then endpos := length (tempword)
{}   else dec(endpos);
{} tempword := copy (tempword,1,endpos);
{}
{} with stringseq do
{}   if postendpos+spaces <= length(str)
{}     then inc(stringseq.pos,endpos+spaces)
{}     else stringseq.pos := 1;
{}
{} GetWord := tempword;
{}End(Function);

var
  Bird_call_descriptor, Current_Process,AccessLog,AccessMap,new_BCD: text;
  matchedlinenumber, line: word;
  matchfound:boolean;
  commandline, BCDline:seqrec;
  currentprocess,fieldname:string[30];
  dirstring:string[10];
  typestring:string[20];
  valuestring,newstring,tempstring:string;
  valerror:integer;

begin
  assign (Bird_call_descriptor, 'BCD.DAT');
  If FileQuery (Bird_call_descriptor, DontAsk) <> Exists
    then runerror (UserMessage('Bird_call_descriptor file (BCD.DAT) not found'));
  reset (Bird_call_descriptor);

  assign (Current_process, 'CRP.DAT');
  If FileQuery (Current_process, DontAsk) <> Exists
    then runerror (UserMessage('Current_process file (CRP.DAT) not found'));
  reset (Current_process);
  readln (Current_process, currentprocess);
  currentprocess := UpString (currentprocess);
  close (Current_process);

  assign (AccessMap, 'BCDACCES.MAP');
  If FileQuery (AccessMap, DontAsk) <> Exists
    then runerror (UserMessage('Access map file (BCDACCES.MAP) not found'));
  reset (AccessMap);

  commandline.str := UpString(command);
  commandline.pos := 1;

  dirstring := GetWord(commandline);
  if (dirstring <> 'READ') and (dirstring <> 'WRITE')
    then runerror (UserMessage('First BCD command must be READ or WRITE, not '+
      dirstring + '.'));

  matchedlinenumber := 0;
  fieldname := GetWord(commandline);
  While not (
    eof (Bird_call_descriptor) or
    (commandline.pos = 1) or
    (fieldname[length(fieldname)] = ':')
  ) do
begin
  matchfound := FALSE;
  While not (eof(Bird_call_descriptor) or matchfound) do
begin
  readln(Bird_call_descriptor,BCDline.str);
  BCDline.pos := 1;
  if UpString(GetWord(BCDline)) = fieldname then matchfound := TRUE;
  inc (matchedlinenumber);
End(While);
  if matchfound then fieldname := GetWord(commandline);
End(While);

if eof(Bird_call_descriptor) then
  runerror (UserMessage('Could not find '+fieldname+' in BCD.DAT'));

if commandline.pos = 1 then
  runerror (UserMessage('No field ending in ":" specified in BCD command: '+command));

```

```

(If the program reaches this point, a field identifier ending with a colon
has been read from the command line, and the BCD file pointer is pointing
after the field headings specified in the command line.)

matchfound := FALSE;
While not (eof(AccessMap) or matchfound) do
begin
  readln(AccessMap,BCDline.str);
  BCDline.pos := 1;
  if UpString(GetWord(BCDline)) = fieldname then matchfound := TRUE;
End{While};

if not matchfound then
  runerror (UserMessage('A "'+fieldname+" was not found in BCDACCESS.MAP'));

typestring := UpString(GetWord(BCDline));

tempstring := '';
matchfound := FALSE;
While not (eof(Bird_call_descriptor) or (tempstring = 'END') or matchfound) do
begin
  readln(Bird_call_descriptor,BCDline.str);
  BCDline.pos := 1;
  tempstring := UpString(GetWord(BCDline));
  if tempstring = fieldname then matchfound := TRUE;
  inc (matchedlinenumber);
End{While};

If not matchfound then
begin
  if tempstring = 'END'
  then tempstring := 'An END was reached in BCD.DAT'
  else tempstring := 'The end of BCD.DAT was reached';
  runerror (UserMessage( tempstring+' before "'+fieldname+" was found'));
End{If};

(If the program reaches this point, a match has been found.
'matchedlinenumber' contains the number of lines which must be read past
to reach the line with the matching field name in BCD.DAT. The current
position of 'BCDline' is pointing at the second word, the field value.
'typestring' contains the field type)

valuestring := GetWord(BCDline);
(valuestring is the entire remainder of the line);

readln (AccessMap, BCDline.str);
BCDline.str := UpString (BCDline.str);
BCDline.pos := 1;
if GetWord(BCDline) <> 'WRITERS:'
  then runerror (UserMessage('The line in BCDACCES.MAP after '+fieldname+' <type> '+
    '<value> should be '+#13+#10+
    'WRITERS: <processname>'+ ' <processname> ...'));
If dirstring = 'READ' then
begin
  readln (AccessMap,BCDline.str);
  BCDline.str := UpString (BCDline.str);
  BCDline.pos := 1;
  if GetWord(BCDline) <> 'READERS:'
  then runerror (UserMessage('The line in BCDACCES.MAP after: '+fieldname+
    ' <type> <value> should be:'+#13+#10+
    'READERS: <processname>'+ ' <processname> ...'));
End{If};

close (AccessMap);

(If the program reaches this point, the current position of BCDline is
pointing at the word following 'WRITERS:' or 'READERS:', depending on the
value of dirstring)

matchfound := FALSE;
Repeat
  if GetWord(BCDline) = currentprocess then matchfound := TRUE;
Until (BCDline.pos = 1) or matchfound;
if not matchfound
  then runerror (UserMessage( currentprocess+' not authorised to '+dirstring+' '+
    copy (fieldname,1,length(fieldname)-1)));

if GetWord (commandline) <> typestring
  then runerror (UserMessage('Type mismatch - note '+fieldname+' '+typestring));

```

```

If dirstring = 'READ' then
begin
  if typestring = 'REAL'
  then val(valuestring, real(value), valerror)
  else if typestring = 'INTEGER'
  then val(valuestring, integer(value), valerror)
  else if typestring = 'WORD'
  then val(valuestring, word(value), valerror)
  else if typestring = 'BYTE'
  then val(valuestring, byte(value), valerror)
  else if typestring = 'LONGINT'
  then val(valuestring, longint(value), valerror)
  else if typestring = 'STRING'
  then string(value) := valuestring
  else runerror (UserMessage( typestring+' not a valid BCD type'));

  if valerror <> 0 then
    runerror (UserMessage('The format of the '+copy (fieldname,1,
      length(fieldname)-1)+' value in BCD.DAT '+'
      'does not match its type'));

  close (Bird_call_descriptor);
End{If};

If dirstring = 'WRITE' then
begin
  if typestring = 'REAL'
  then str(real(value), valuestring)
  else if typestring = 'INTEGER'
  then str(integer(value), valuestring)
  else if typestring = 'WORD'
  then str(word(value), valuestring)
  else if typestring = 'BYTE'
  then str(byte(value), valuestring)
  else if typestring = 'LONGINT'
  then str(longint(value), valuestring)
  else if typestring = 'STRING'
  then valuestring := string(value)
  else runerror (UserMessage(typestring+' not a valid BCD type'));

  assign (new_BCD, 'BCDTEMP.DAT');
  rewrite (new_BCD);
  reset (Bird_call_descriptor);

  For line := 1 to matchedlinenumber-1 do
  begin
    readln (Bird_call_descriptor, tempstring);
    writeln (new_BCD, tempstring);
  End{For};

  readln (Bird_call_descriptor, BCDline.str);
  BCDline.pos := 1;
  tempstring := GetWord(BCDline); {point past the field identifier}
  with BCDline do if pos = 1
  then newstring := str+' '+valuestring
  else begin
    newstring := copy(str,1,pos-1)+' '+valuestring;
    tempstring := GetWord(BCDline); {skip past old value}
    with BCDline do if pos <> 1
    then newstring := newstring+copy(str,pos,length(str)-pos+1);
  end;

  {the above is done to preserve the identifier case, indentation and
  comments such as unit values in BCD.DAT, although this is not necessary
  for the correct operation of BCDMANIP}
  writeln (new_BCD, newstring);

  While not eof(Bird_call_descriptor) do
  begin
    readln (Bird_call_descriptor, tempstring);
    writeln (new_BCD, tempstring);
  End{While};

  close (Bird_call_descriptor);
  close (new_BCD);
  Erase (Bird_call_descriptor);
  Rename (new_BCD, 'BCD.DAT');
End{If};
End{Procedure BCD};

```

```

Procedure PutCurrentProcess (process:string);
var
    Current_process:text;
begin
    assign (Current_process,'CRP.DAT');
    rewrite (Current_process);
    writeln (Current_process, UpString(process));
    close (Current_process);
End(Procedure);

Procedure GetCurrentProcess (var process:string);
var
    Current_process:text;
begin
    assign (Current_process,'CRP.DAT');
    reset (Current_process);
    readln (Current_process, process);
    close (Current_process);
End(Procedure);

END{UNIT}.

```

B1.17 GENLIB.PAS

```
UNIT GENLIB; {General library routines}
INTERFACE
USES Dos;

TYPE
  Long = record           {used to access the high and}
    Lo,Hi: word; {low words of a 4-byte type}
  end;

TYPE
  queryresult = (OverWriteOK, DontOverWrite,
    DoesNotExist,Exists, Unassigned);
  asktype = (AskOverWrite, DontAsk);

Function FileQuery (var filevar; ask:asktype): queryresult;
{
  IF [no error other than 'file not found' occurs when opening 'filevar']
  THEN IF ['filevar' exists in the current directory]
    THEN IF ['ask' requires the routine to ask the user about overwriting]
      THEN [place message at bottom of screen asking whether it is OK to
        overwrite the file. Return 'OverWriteOK' or 'DontOverWrite'
        according to the response]
      ENDIF
    ELSE [return 'Exists']
  ELSE [return 'DoesNotExist']
  ENDIF
ELSE [allow the normal run-time error to occur]
ENDIF
}
{
  typed, untyped and text files are all handled. If a non file variable is
  passed 'Run time error 102: File not assigned' occurs
}

Procedure GraphTransform (var InPoints; NumInPoints:word;
  var OutPoints; NumOutPoints:word);
{
  [Perform a linear interpolation on an array ('InPoints') of 'NumInPoints'
  real values, then pick out 'NumOutPoints' evenly-spaced values on the
  interpolated curve and return them in the 'OutPoints' array]
}
{
  The variables passed via 'InPoints' and 'OutPoints' can be of any size and
  type, but are assumed to be arrays of size 'NumInPoints' and 'NumOutPoints'
  respectively, and of type REAL.
}

Function Mean (var InPoints; NumInPoints: word;
  var Weights; WeightFlag: boolean): real;
{
  Return the mean of the values in InPoints, weighted by the values in
  Weights if WeightFlag is true.
}
{
  The variables passed via 'InPoints' and 'Weights' can be of any size and
  type, but are assumed to be arrays of size 'NumInPoints' and of type REAL.
}

Function StdDev (var InPoints; Mean: real; NumInPoints: word;
  var Weights; WeightFlag:boolean):real;
{
  Return the root of the second moment of the values in 'InPoints' about the
  'Mean' value. If 'Mean' is the mean of the 'InPoints' data, the returned
  result is the standard deviation of that data. The moment is weighted by
  the values in 'Weights' if 'WeightFlag' is true.
}
{
  The variables passed via 'InPoints' and 'Weights' can be of any size and
  type, but are assumed to be arrays of size 'NumInPoints' and of type REAL.
}

Procedure GenRunError (errornumber:byte; errormessage:string);
{
  [write the 'errormessage' string to the file RUNERROR.MSG and halt
  execution with Run Time Error no. 'errornumber']
}
```

```

Procedure WriteXY (x,y: byte; mess: string);
{
  Writes the 'mess' string at text coords (x,y), leaving the cursor position
  unchanged. If the coord is 0, the current coord is assumed.
}

Function UserMessage (mess: string): byte;
{
  Writes out the 'mess' string (using BIOS scrren access so the characters
  appear on graphics screens too) and returns the value 255, which is an unused
  runtime error number. When called as follows:

      runerror(UserMessage('User Error'));

  will cause "User Message" to be written followed by "Runtime error 255 at
  <Seg>:<Ofs>" and if in the integrated environment, the cursor will be placed
  at the error position.
}

Function AbsAddr(ptr:pointer): longint;
{
  Returns the absolute (24-BIT) address of the location
  pointed to by 'ptr'
}

Function AbsPtr (absadd:longint): pointer;
{
  returns a pointer pointing to the location at the absolute (24-BIT)
  address specified by 'absadd'. The pointer is normalised so that the
  segment part points to the start of a 64k block of memory, ie it is
  of the form s000:0000 where s is 0..9.
}

Function UpString (instring:string):string;
{
  Convert the input string 'instring' to uppercase.
}

Function Hex (decvalue:longint):string;
{
  Convert the input integer to a hexadecimal string.
}

Function UnHex (S:string): longint;
{
  Convert string in hexadecimal to integer. Skips over non hex characters.
  Converts from LS digit to MS digit and stops before the hex digit that would
  cause an overflow.
}

Function Strung (field, decplaces: byte; number: real): string;
{
  [return the string of a number, in a field of 'field' characters
   and with 'decplaces' digits after the decimal point]
  Just put decplaces=0 for integers.
}

Function FitsSpec (Path,Spec: PathStr): boolean;
{
  Return TRUE if the DOS file name passed in 'Name' fits the general
  dos spec passed in 'Spec'.
}

Procedure Showfile (FileName: PathStr);
{
  Displays a text file.
}

Procedure DosRead (var s: string);
{
  Uses the dos buffered input function to read a string from standard input
  device. The DOS line editing keys, or programs like DOSEDIT are then active.
}

(=====OBJECTS====)

```

{-----STRINGSEQ object--}

TYPE

```
StringSeq = object
    pos: byte;
    str: string;
    EndOfSequence: boolean;
    whitespaceset: set of char;
    whitespacestr: string[2];

    constructor Init (s: string);
    function GetWord: string;
    function Rest: string;
    procedure Reset;
    destructor Done;
end;
```

{
STRINGSEQ OBJECT

A sequence of character groups (words) separated by spaces. The words can be read in sequence like numbers in a TEXT file, by subsequent calls to a read routine.

STR

String contains the sequence

POS

A pointer to a character inside STR (usually the next non-space character to be read).

WHITESPACESTR, WHITESPACESET

Define the set of characters regarded as the space between words. If this is to change, either redefine whitespacestr (whitespaceset is automatically generated) remembering to change the length of the string, or make the string specifiable as a parameter to INIT.

INIT

Used to supply the sequence value, and to create dynamically.

GETWORD

Returns the next word in the sequence

REST

Returns the entire string from the current position to the end and resets the sequence

RESET

Sets the current reading position to the start of the sequence

ENDOFSEQUENCE

Returns TRUE if there are no more words to read.

DONE

Used to dispose dynamically.

}

{-----TextLines OBJECT--}

TYPE

```
TextLines = object
    LinesRead: longint;
    TheFile: text;

    constructor Open (filename: string);
    function TheFileName: string;
    function NextLine: string;
    destructor Close;
End; {TextLines}
```

{

OBJECT TEXTLINES

A text file that is meant to be read in a linewise fashion.

LINESREAD

Contains the number of lines read since the last Open or Reset.

THEFILENAME

The name (without the path) of the file.

THEFILE

The text file variable.

OPEN

Opens the file 'filename' for reading if possible. IOresult will contain the relevant value if (\$I-) was used. If filename is %%Reopen%% then resets an already open file.

```

NEXTLINE
Does a readln and returns the resulting string. If the file was at EOF,
returns the string '%%End Of File%%'.

CLOSE
Closes the file.
}
{=====END OF OBJECTS==}

IMPLEMENTATION
USES Crt;

Function FileQuery;
var
  io: word;
  ch: char;
  name: string[80];
  j: 0..79;
  filewasclosed, fileistext, fileunassigned: boolean;
begin
  filewasclosed := FALSE;
  fileistext := FALSE;
  fileunassigned := FALSE;
  if filerec(filevar).Mode = $D7B0 then filewasclosed := TRUE
  else if (filerec(filevar).Mode = $D7B1) or
    (filerec(filevar).Mode = $D7B2) then fileistext := TRUE
  else if filerec(filevar).Mode <> $D7B3 then fileunassigned := TRUE;

  If not fileunassigned
  then begin
    {I-}
    if filewasclosed then if fileistext
      then reset(text(filevar))
      else reset(file(filevar))
    else if fileistext
      then append(text(filevar))
      else seek (file(filevar),filesize(file(filevar)));
    {I+}

    io := IOresult;
    if io in [0,2]
    then if io = 0
      then Begin
        if ask = AskOverWrite
        then Begin
          sound (300); delay (500); nosound;
          name := '';
          j := 0;
          Repeat
            name:=name+filerec(filevar).name[j];
            inc(j);
          Until (name[j]=#0);

          write ('The file '+name+' already exists.'+
            ' Overwrite? (Y/N) ');
          ch := readkey;
          if Upcase(ch) = 'Y' then FileQuery := OverWriteOK
            else FileQuery := DontOverWrite;
          writeln (ch);
        End
        else FileQuery := Exists;
        if filewasclosed then close(file(filevar));
      End
      else FileQuery := DoesNotExist
    else if fileistext then reset (text(filevar)) else reset(file(filevar))
    {A file error other than 'File not found' occurs}
  end
else FileQuery := Unassigned;

End{Function};

Procedure GraphTransform;
type RealArray = array [1..$FFFF div sizeof(real)] of real; {largest real
array possible}
var
  Ratio: Real; {Ratio of source points to screen points}
  X: Word; {The screen point whose Y-value is currently being calculated}
  BetweenPnt: Real; {"Ghost" point between actual source points corres-}
  LeftPnt: Word; {ponding to the exact position of the screen point}
  LeftPntVal, {The actual source point closest to the ghost}
  {point on the side closer to the origin}
  {The value at 'LeftPnt'}

```

```

    RightPntVal: Real;           {The value at the source point closest to the}
                                {ghost point on the side away from the origin}
begin
    Ratio := NumInPoints/NumOutPoints;
    BetweenPnt := 1;
    For X := 1 to NumOutPoints do
    begin
        BetweenPnt := BetweenPnt+Ratio;
        LeftPnt := Trunc (BetweenPnt);
        LeftPntVal := RealArray (InPoints)[LeftPnt];
        RightPntVal := RealArray (InPoints)[LeftPnt + 1];
        RealArray (OutPoints) [X] := {The Y value}
            LeftPntVal + Frac(BetweenPnt) * (RightPntVal-LeftPntVal);
    end{For};
end{Procedure};

Function Mean (var InPoints; NumInpoints: word;
               var Weights; WeightFlag: boolean): real;
type
    RealArray = array [1..$FFFF div sizeof(real)] of real; {largest real
                                                             array possible}
var
    total, WeightSum: real;
    i: word;
begin
    total := 0;
    if WeightFlag
    then begin
        WeightSum := 0;
        for i := 1 to NumInPoints do
            WeightSum := WeightSum + realarray(Weights)[i];
        for i := 1 to NumInPoints do
            total := total+realarray(InPoints)[i]*realarray(Weights)[i];
        if weightsum <> 0 then Mean := total/WeightSum else Mean := 0;
        end
    else begin
        for i := 1 to NumInPoints do total := total+realarray(InPoints)[i];
        Mean := total/NumInpoints;
        end;
    end;
end;{Function Mean}

Function StdDev (var InPoints; Mean: real; NumInpoints: word;
                 var Weights; WeightFlag: boolean): real;
type
    RealArray = array [1..$FFFF div sizeof(real)] of real; {largest real
                                                             array possible}
var
    total, WeightSum: real;
    i: word;
begin
    total := 0;
    if WeightFlag
    then begin
        WeightSum := 0;
        for i := 1 to NumInPoints do
            WeightSum := WeightSum + realarray(Weights)[i];
        for i := 1 to NumInPoints do
            total := total+sqr(realarray(InPoints)[i]-Mean)*realarray(Weights)[i];
        if WeightSum<>0 then StdDev := sqrt(total/WeightSum) else StdDev := 0;
        end
    else begin
        for i := 1 to NumInPoints do
            total := total+sqr(realarray(InPoints)[i]-Mean);
        StdDev := sqrt(total/NumInpoints);
        end;
    end;
end;{Function StdDev}

Procedure GenRunError (errornumber:byte; errormessage:string);
begin
    directvideo := FALSE;
    writeln (errormessage);
    directvideo := TRUE;
    halt(errornumber);
end{Procedure};

Procedure WriteXY;
var
    ox,oy: byte;
begin
    ox := whereX;
    oy := whereY;
    if x = 0 then x := ox;
    if y = 0 then y := oy;

```

```

    gotoxy (x,y);
    write (mess);
    gotoxy (ox,oy);
End;

Function UserMessage;
begin
    directvideo := FALSE;
    writeln (mess);
    UserMessage := 255;
End;

function AbsAddr(ptr:pointer): longint;
{
    returns the absolute (24-BIT) address of the location
    pointed to by 'ptr'
}
var tempaddr: longint;
begin
    tempaddr := Long(ptr).Hi;           {This ensures longint}
    tempaddr := tempaddr shl 4 + Long(ptr).Lo; {arithmetic is used}
    AbsAddr := tempaddr;
end;

function AbsPtr (absadd:longint): pointer;
{
    returns a pointer pointing to the location at the absolute (24-BIT)
    address specified by 'absadd'. The pointer is normalised so that the
    segment part points to the start of a 64k block of memory, ie it is
    of the form s000:0000 where s is 0..9.
}
var tempptr:pointer;
begin
    Long(tempptr).Hi := Long(absadd).Hi*4096;
    Long(tempptr).Lo := Long(absadd).Lo;
    AbsPtr := tempptr;
end;

Function UpString (instring:string):string;
{
    [convert the input string 'instring' to uppercase]
}
var i:byte;
begin
    for i := 1 to length(instring) do instring[i] := upcase (instring[i]);
    UpString := instring;
End{Function};

(*
Function Hex (decvalue:longint):string;
begin
    if decvalue = -maxlongint then Hex := '8000 0000';
    if decvalue < 0 then Hex := '-' + Hex(not(decvalue)+1);
    if decvalue > 65535
    then Hex := Hex(decvalue div 65536) + ' ' +
        Hex(hi(decvalue mod 65536) div 16) +
        Hex(hi(decvalue mod 65536) mod 16) +
        Hex(lo(decvalue mod 65536) div 16) +
        Hex(lo(decvalue mod 65536) mod 16)
    else if decvalue > 255
    then Hex := Hex(hi(decvalue)) +
        Hex(lo(decvalue) div 16) +
        Hex(lo(decvalue) mod 16)
    else if decvalue > 15
    then Hex := Hex(decvalue div 16) + Hex(decvalue mod 16)
    else if decvalue > 9
    then Hex := char (55+decvalue)
    else Hex := char (48+decvalue);
End{Function};
*)

Function Hex;

{}function character (c:byte): char;
{}begin
{}    if c > 9
{}    then character := chr (c+ord('A')-10)
{}    else character := chr (c+ord('0'));
{}end;

var
    temp: string;

```

```

begin
  temp := '';
  repeat
    temp := character(decvalue and $F) + temp;
    decvalue := decvalue shr 4;
  until (decvalue = 0);
  Hex := temp;
End {Function Hex};

Function UnHex;
var
  i, shift: byte;
  tempresult, digit: longint;
  overflowimminent: boolean;

{}procedure dorest(digit:longint);
{}begin
{}  digit := digit shl shift;
{}  inc(tempresult, digit);
{}  if digit < 0 then overflowimminent := TRUE;
{}  inc(shift, 4);
{}end;

Begin
  shift := 0;
  i := length(S);
  tempresult := 0;
  overflowimminent := FALSE;
  While (not overflowimminent) and (i > 0) do
  begin
    if S[i] in ['0'..'9']
    then dorest(ord(S[i])-ord('0'))
    else begin
      S[i] := UpCase(S[i]);
      if S[i] in ['A'..'F']
      then dorest(ord(S[i])-ord('A')+10)
      end;
    dec(i);
  End{While};
  UnHex := tempresult;
End {Function UnHex};

Function Strung (field, decplaces: byte; number: real): string;
{
  [return the string of a real number, in a field of 'field' characters
   and with 'decplaces' digits after the decimal point]
}
var strg: string;
begin
  str (number:field:decplaces, strg);
  Strung := strg;
End{Function Strung};

Function FitsSpec (Path, Spec: PathStr): boolean;
var
  Dir, SpecDir: DirStr;
  Name, SpecName: NameStr;
  Ext, SpecExt: ExtStr;

{}Function WildCardFit (N,S: NameStr): boolean;
{}var
{}  i, starpos: byte;
{}begin
{}  WildCardFit := TRUE;
{}  starpos := pos ('*', S);
{}  if starpos > 0
{}  then begin
{}    N := copy(N, 1, starpos-1);
{}    S := copy (S, 1, starpos-1);
{}    end;
{}  if length (S) <> length (N)
{}  then WildCardFit := FALSE
{}  else for i := 1 to length(N) do
{}    if not (S[i] in [N[i], '?']) then WildCardFit := FALSE;
{}  End;

begin
  FitsSpec := TRUE;
  fsplit (UpString(Path), Dir, Name, Ext);
  fsplit (UpString(Spec), SpecDir, SpecName, SpecExt);
  if Dir <> SpecDir
  then FitsSpec:= FALSE
  else if not WildCardFit (Name, SpecName)
  then FitsSpec:= FALSE

```

```

        else if not WildCardFit(Ext,SpecExt)
            then FitsSpec:= FALSE;
End; {FitsSpec}

Procedure Showfile (FileName: PathStr);
var
    f: text;
    line: string[80];
    linecount: word;
begin
    assign (f,FileName);
    {$I-} reset(f); {$I+}
    If IOresult = 0
    then begin
        linecount := 0;
        while not eof(f) do
            begin
                inc(linecount);
                if linecount mod 25 = 0
                then begin
                    writexy (WhereX,WhereY,'***** Press a key to continue *****');
                    line := readkey;
                    writexy (WhereX,WhereY,'
                ');
                end
                else begin
                    readln(f,line);
                    writeln(line);
                end;
            end;
        close(f);
    end
    else writeln ('Error opening ',FileName);
End;

Procedure DosRead (var s: string);
var
    regs: registers;
begin
    regs.ah := $A;
    regs.ds := Seg(s);
    regs.dx := ofs(s);
    s[0] := #255;
    intr($21,regs);
    s[0] := char(ord(s[1])+1);
    s := copy (s,2,254);
    writeln;
End {DosRead};

{=====METHODS=====}

{-----STRINGSEQ methods-----}
Constructor StringSeq.Init (s:string);
var i: byte;
begin
    str := s;
    whitespacestr := #32#9;
    whitespaceset := [];
    for i := 1 to length (whitespacestr) do
        whitespaceset := whitespaceset + [whitespacestr[i]];
    Reset;
End;

Function StringSeq.GetWord: string;
var
    endpos, tempendpos, spaces:byte;
    tempword:string;
    i: byte;
begin
    If not EndOfSequence
    then begin
        tempword := copy (str,pos,length(str)-pos+1);

        spaces := length (tempword);
        while (tempword[1] in whitespaceset) and (length(tempword) > 0) do
            tempword := copy(tempword,2,length(tempword)-1);
            dec(spaces,length(tempword));
    end

```

```

endpos := 0;
for i := 1 to length(whitespacestr) do
begin
    tempendpos := system.pos (whitespacestr[i],tempword);
    if tempendpos <> 0 then
        if endpos = 0
            then endpos := tempendpos
            else if endpos > tempendpos then endpos := tempendpos;
    end;

    if endpos = 0 then endpos := length (tempword)
        else dec(endpos);
    tempword := copy (tempword,1,endpos);

    if pos+endpos+spaces < length(str)
        then inc(pos,endpos+spaces)
        else EndOfSequence := TRUE;

    GetWord := tempword;
end
else GetWord := '';
End;

Function StringSeq.Rest: string;
begin
    if not EndOfSequence
    then begin
        Rest := copy (str,pos,length(str)-pos+1);
        Reset;
    end
    else Rest := '';
End;

Procedure StringSeq.Reset;
begin
    EndOfSequence := TRUE;
    if length(str) <> 0
    then begin
        pos := 1;
        while str[pos] = ' ' do inc(pos);
        if pos < length (str) then EndOfSequence := FALSE;
        pos := 1;
    end;
End;

Destructor StringSeq.Done;
begin
End;

{-----TextLines METHODS---}
Constructor TextLines.Open;
begin
    LinesRead := 0;
    If filename <> '%%Reopen%%' then assign (TheFile, filename);
    {$I-} reset (TheFile); {$I+}
    if IOResult <> 0 then runerror(UserMessage('Error opening '+filename));
End; {TextLines.Open}

Function TextLines.TheFileName;
begin
    TheFileName := copy(textrec(TheFile).Name,1,pos(#0,textrec(TheFile).Name)-1);
End;

Function TextLines.NextLine;
var
    tempNextLine: string;
begin
    if not eof(TheFile)
    then begin
        readln (TheFile,tempNextLine);
        inc (LinesRead);
        NextLine := tempNextLine;
    end
    else NextLine := '%%End Of File%%';
End; {TextLines.NextLine}

Destructor TextLines.Close;
begin
    system.close (TheFile);
End; {TextLines.Close}

```

```
{=====END OF METHODS==}
```

```
END.{UNIT}
```

B1.18 OBJLIB.PAS

```
UNIT OBJLIB; {Library of general purpose objects}

INTERFACE

{=====ListItem OBJECT==}
TYPE
  ListItemPtr = ^ListItem;
  ListItem = Object
    Next: ListItemPtr;

    constructor Init;
    destructor Done; virtual;
End; {ListItem}

{=====List OBJECT==}
TYPE
  List = Object
    Head,Tail,Current: ListItemPtr;

    constructor Init;
    destructor Done (ZapNodes: boolean);
    procedure Attach (P:ListItemPtr);
    procedure SetToHead;
    procedure SetToNext;
    function Item (Detach:boolean): ListItemPtr;
    function AtTail: boolean;
    function Empty: boolean;
End; {List}

{
OBJECT LIST
A forward linked list with a dummy node for both head and tail, and
an implicit current position. Attach and detach nodes in a FIFO stack-like
manner, but from any point in the list.

HEAD
Points to the head node.

TAIL
Points to the tail. The tail's link points to NIL.

CURRENT
Points to the current node - this can be Head or Tail too. Defaults to Head.

ATTAIL
True when the current position is the tail.

SETTOHEAD
Set the current pointer to the head.

EMPTY
True when the list is empty, ie there are only head and tail nodes - the
head is linked to the tail.

ITEM
Item (FALSE) returns a pointer to the current node, or NIL if that is the
head or tail. Item (TRUE) does the same, and then detaches the node from
the list. By overriding Item to return a new item type as follows...

  type MyItem = object (ListItem)
    newfield: integer;
    destructor Done; virtual;
  end;
  MyItemPtr = ^MyItem;

  type MyListType = object (List)
    function Item (Detach: boolean): MyItemPtr;
  end;
  function MyListType.Item
begin
  Item = MyItemPtr(List.Item(Detach));
End;

... you can do the following:

var
  MyList: MyListType;

begin
  MyList.Item(FALSE)^.Newfield
dispose (MyList.Item(TRUE),Done);
end.
```

In other words, you can access new fields without having to typecast the pointer that Item returns, and dispose will call the new MyItem.Done.

When using Item to detach nodes, note that the current pointer will be left pointing at the next node towards the tail. Nothing happens when Current is Head or Tail.

SETTONEXT

Sets the current pointer to the item following the current node. If the current node is the tail, leaves it there.

ATTACH

Links the node whose pointer is passed in (P). The node is attached at the position after the current pointer, leaving Current at the new node. If the requested link is impossible, (ie Current points to Tail), nothing happens.

The above means items can be attached in a towards-Tail direction starting anywhere in the list, by repeated calls to Attach. Eg to attach to the start of the list, SetToHead (default) and repeatedly Attach.

INIT

Creates an empty list with the current position being the head. Note that the list can be used either dynamically or statically (you pass the node pointers, so you decide where they come from) but it will always create the head and tail nodes on the heap.

DONE

Disposes list and/or does Garbage collection (disposes head and tail nodes). If parameter passed (ZapNodes) is TRUE, disposes the node items, calling their Done methods. (Note the Dones are virtual).

```
}

{=====DListItem OBJECT===}
TYPE
  DListItemPtr = ^DListItem;
  DListItem = object (ListItem)
    Prev: DListItemPtr;
  end;

{=====Dlist OBJECT===}
  DList = object (List)
    Reverse: boolean;

    constructor Init;
    procedure Attach (P:DListItemPtr);
    function AtHead: boolean;
    procedure SetToTail;
    procedure SetToPrev;
    procedure SetReverse (D:boolean);
    function Item (Detach:boolean): DListItemPtr;
  end;
{
  OBJECT DLIST
  Like List, but doubly-linked - ie both forward and backward linked.

  REVERSE
  Direction for attaching and detaching. Defaults to FALSE.

  INIT
  Calls List's Init and initialises Head and Tail's Prev fields. (Head's Prev
  is NIL).

  SETREVERSE
  Sets reverse to value of passed-in parameter (D). Set FALSE for attaching
  and detaching like with List, TRUE for the opposite directions.

  ATTACH
  If Reverse is FALSE, behaves like List's Attach. If TRUE, it attaches
  the node at position before Current, moving Current to the new node. This
  means attaching in a towards-Head direction. If Current is the head, nothing
  happens.

  ATHEAD
  Complement of AtTail.

  SETTOTAIL
  Complement of SetToHead.

  SETTOPREV
  Complement of SetToNext.
```

```

ITEM
When Reverse is FALSE, identical in function to List's Item - but works
faster by using the Prev field. When Reverse is TRUE, leaves the current
pointer at the previous position when detaching. Returns a DListItemPtr
rather than a ListItemPtr.
}

{=====Stack OBJECT==}
TYPE
  Stack = object (List)
    procedure Push (P: ListItemPtr);
    procedure Pop (var P: ListItemPtr);
  End; {Stack}

{
  OBJECT STACK
  A FIFO stack - access using only Push and Pop. In fact the Empty function
  might be useful as well, and all the List operations are available in
  principle.

  PUSH
  Add the item whose pointer is passed in (P) to the top of the list.

  POP
  Detach the first list item and return its pointer. Return NIL if the stack
  is empty.
}

{=====NumberedList OBJECT==}
{
  Ordinary list, but maintains a count, 'NumberOfItems', of the items.
}
{=====}
TYPE
  NumberedList = object (List)
    NumberOfItems: word;

    constructor Init;
    procedure Attach (P: ListItemPtr);
    function Item (Detach: boolean): ListItemPtr;
  End; {NumberedList}

IMPLEMENTATION
USES GENLIB;
{=====ListItem METHODS==}
  Constructor ListItem.Init;
  begin
  End; {ListItem.Init}

  Destructor ListItem.Done;
  begin
  End; {ListItem.Done}

{=====List METHODS==}
  Constructor List.Init;
  begin
    new (Head, Init);
    new (Tail, Init);
    Head^.Next := Tail;
    Tail^.Next := NIL;
    Current := Head;
  End; {List.Init}

  Procedure List.SetToHead;
  begin
    Current := Head;
  End; {List.SetToHead}

  Procedure List.SetToNext;
  begin
    if not (Current = Tail) then Current := Current^.Next;
  End; {List.SetToNext}

  Function List.Item;
  var
    Prev: ListItemPtr;
  begin
    if (Current = Head) or (Current = Tail)
    then Item := NIL
    else begin
      Item := Current;

```

```

        If Detach then
        begin
            Prev := Head;
            while Prev^.Next <> Current do Prev := Prev^.Next;
            Prev^.Next := Current^.Next;
            Current := Current^.Next;
        end; {If Detach...}
    end;
End; {List.Item}

Function List.AtTail;
begin
    AtTail := (Current = Tail);
End; {List.AtTail}

Function List.Empty;
begin
    Empty := (Head^.Next = Tail);
End; {List.Empty}

Procedure List.Attach;
begin
    if Current <> Tail
    then begin
        P^.Next := Current^.Next;
        Current^.Next := P;
        Current := P;
    end
    else runerror(UserMessage('Attach to Tail failed, P=' +
        Hex(Seg(P^)) + ' : ' + Hex(Offs(P^))))
End; {List.Attach}

Destructor List.Done;
var Temp: ListItemPtr;
begin
    Current := Head;
    repeat
        Temp := Current;
        Current := Current^.Next;
        if ZapNodes or (Temp = Head)
        then dispose (Temp,Done);
    until Current = Tail;
    dispose (Tail,Done);
End; {List.Done}

{=====NumberedList METHODS==}
Constructor NumberedList.Init;
begin
    List.Init;
    NumberOfItems := 0;
End; {NumberedList.Init}

Procedure NumberedList.Attach;
begin
    List.Attach(P);
    inc (NumberOfItems);
End; {NumberedList.Attach}

Function NumberedList.Item;
begin
    Item := List.Item(Detach);
    if Detach then dec (NumberOfItems);
End; {NumberedList.Item}

{=====DList METHODS==}
Constructor DList.Init;
begin
    {Copy of List.Init, but using DListItemPtr's}
    new (DListItemPtr(Head),Init);
    new (DListItemPtr(Tail),Init);
    Head^.Next := Tail;
    Tail^.Next := NIL;
    Current := Head;

    {New stuff}
    DListItemPtr(Head)^.Prev := NIL;
    DListItemPtr(Tail)^.Prev := DListItemPtr(Head);
    Reverse := FALSE;
End; {DList.Init}

```

```

Procedure DList.SetToTail;
begin
  Current := Tail;
End; {DList.SetToTail}

Procedure DList.SetToPrev;
begin
  if not (Current = Head) then Current := DlistItemPtr(Current)^.Prev;
End; {DList.SetToPrev}

Procedure DList.SetReverse;
begin
  Reverse := D;
End; {DList.SetReverse}

Function DList.AtHead;
begin
  AtHead := (Current = Head);
End; {DList.AtHead}

Function DList.Item;
begin
  if (Current = Head) or (Current = Tail)
  then Item := NIL
  else begin
    Item := DlistItemPtr(Current);
    If Detach then
      begin
        DlistItemPtr(Current)^.Prev^.Next := Current^.Next;
        DlistItemPtr(Current^.Next)^.Prev := DlistItemPtr(Current)^.Prev;
        If Reverse then Current := DlistItemPtr(Current)^.Prev
          else Current := Current^.Next;
        End; {If Detach...}
      end;
  end;
End; {DList.Item}

Procedure DList.Attach;
var
  Neighbour: DlistItemPtr;
begin
  if not Reverse {insert after current}
  then if Current <> Tail
    then begin
      Neighbour := DlistItemPtr(Current^.Next);
      Current^.Next := P;
      Neighbour^.Prev := P;
      P^.Next := Neighbour;
      P^.Prev := DlistItemPtr(Current);
      Current := P;
    end
    else runerror(UserMessage('Attach to Tail failed, P=' +
      Hex(Seg(P^)) + ':' + Hex(Ofs(P^))))
  else if Current <> Head {insert before current}
  then begin
    Neighbour := DlistItemPtr(Current)^.Prev;
    Neighbour^.Next := P;
    DlistItemPtr(Current)^.Prev := P;
    P^.Next := Current;
    P^.Prev := Neighbour;
    Current := P;
  end
  else runerror(UserMessage('Attach to Head failed, P=' +
    Hex(Seg(P^)) + ':' + Hex(Ofs(P^))))
End; {DList.Attach}

{=====Stack METHODS=====}
Procedure Stack.Push;
begin
  SetToHead;
  Attach(P);
End; {Stack.Push}

Procedure Stack.Pop;
begin
  If Empty
  then P := NIL
  else begin
    SetToHead;
    SetToNext;
    P := Item(TRUE);
  end;
End;

{=====BinaryTree METHODS=====}

```

END.

B1.19 TEXTWIN.PAS

```
UNIT TEXTWIN; {Text window object}
INTERFACE
{=====OBJECTS==}

{=====TState OBJECT==}
TYPE
  TState = object
    x1,y1,x2,y2,FCol,BCol,Cx,Cy: byte;

    procedure SetWindow (x1,y1,x2,y2: byte);
    procedure SetTextAttr (txcol,bkcol:byte);
    procedure SetCursor (x,y: byte);
  End; {TState}

{
  TSTATE
  Object used to manipulate the text state. It is not necessary to construct
  since it is meant to be used only for the TextState object defined by this
  unit, and requires no virtual methods or formal initialisation. It is to
  this state that all windows set the text system when deactivating
  themselves. So use this if you want the relevant parameters to be preserved
  between window activations.

  X1,Y1,X2,Y2
  Parameters of the current coord system.

  FCOL, BCOL
  Current colours (text attributes).

  SETWINDOW
  Use to set the active text window.

  SETTEXTATTR
  Use to set the current foreground and background colours.

  SETCURSOR
  Use to set the current cursor position
}

VAR
  TextState: Tstate; {Use this TState object to change windows or colours if}
                    {you wish these parameters to be correctly preserved }
                    {by in between TextWindow activations.           }

{=====TextWindow OBJECT==}
TYPE
  TextWindow = object
    x1,y1,x2,y2,w,d,FCol,BCol,Cx,Cy: byte;
    Border, Active: boolean;
    OImage: pointer;

    constructor Init (x,y,width,depth,txcol,bkcol: byte;
                      bord: boolean; Title: string);
    procedure Activate;
    procedure Deactivate;
    destructor Done;
  End;

{
  TEXTWINDOW
  A text window.

  X1,Y1,X2,Y2
  Top left and bottom right coord of window.

  W,D
  Width and depth dimensions of window.

  BORDER, ACTIVE
  True when a border has been drawn and when the window is active,
  respectively.

  FCOL, BCOL, CX, CY
  Foreground and background colours of, and cursor position in, this window
  when it was last deactivated.
}
```

```

INIT
Opens the window. The position and dimension fields are set to the parameters
(x,y,width, depth) specified, with the colours (txcol, bkcol) specified.
If txcol = bkcol then the window's attributes are taken from the TextState
variable. The text underneath the window is stored on the heap. The window
is cleared and the current window coords (Turbo Pascal window) set to the
window size. If bord is TRUE, a border is drawn and the window coords set to
an accordingly smaller size. The window is made active.

SETBORDER
Draws a border, and sets Border to true. If 'Title' is not empty, the title
is printed (As much as possible) as the top border. Sets the current window
coords (tTurbo Pascal window) to the area inside the border.

ACTIVATE
Sets the TP window, cursor position and text attributes to the values for
this window.

DEACTIVATE
Sets the TP window, cursor position and text attributes to the values
specified by the TextState global object.

DONE
Deactivates and then closes the window. The screen image that was under the
window is restored.
}

```

```

(=====END OF OBJECTS==)

```

```

IMPLEMENTATION
USES graph, crt;
CONST
    Unchanged = -1;
VAR
    ScreenSeg: word; {Segment of text mode screen, set up by unit initialisation}

Procedure SaveRect (ScreenAddr: pointer;
                    MoveBytes: word;
                    BytesPerLine: word;
                    NoOfLines: word;
                    Destination: pointer);
{
    moves the bytes from ScreenAddr to destination,
    in NoOfLines lines of BytesPerLine bytes each. Only the first
    MoveBytes lines are moved. In practice, if the ScreenAddr points to some
    offset in the line, that offset will be used for all the lines, so a
    rectangle is moved.
}
{
var
    scrnseg, scrnofs: word;
    destseg, destofs: word;
    bytes, lines: byte;
begin
    scrnseg := Seg(ScreenAddr^);
    scrnofs := Ofs(ScreenAddr^);
    destseg := Seg(Destination^);
    destofs := Ofs(Destination^);
    for lines := 1 to NoOfLines do
    begin
        for bytes := 1 to MoveBytes do
        begin
            Mem[destseg: destofs] := Mem[scrnseg: scrnofs];
            inc (scrnofs);
            inc (destofs);
        end;
        inc(scrnofs, BytesPerLine-Movebytes);
    end;
end;
}
External; {$L SAVERECT.OBJ}

```

```

Procedure RestRect (ScreenAddr: pointer;
                    MoveBytes: word;
                    BytesPerLine: word;
                    NoOfLines: word;
                    Source: pointer);
{
    moves the bytes from Source to ScreenAddr,
    in NoOfLines lines of BytesPerLine bytes each. Only the first
    MoveBytes lines are moved. In practice, if the ScreenAddr points to some
    offset in the line, that offset will be used for all the lines, so a
    rectangle is moved.
}

```

```

}
{
var
  scrnseg, scrnofs: word;
  srceseg, srceofs: word;
  bytes, lines: byte;
begin
  scrnseg := Seg(ScreenAddr^);
  scrnofs := Ofs(ScreenAddr^);
  srceseg := Seg(Source^);
  srceofs := Ofs(Source^);
  for lines := 1 to NoOfLines do
  begin
    for bytes := 1 to MoveBytes do
    begin
      Mem[scrnseg: scrnofs] := Mem[srceseg:srceofs];
      inc(scrnofs);
      inc (srceofs);
    end;
    inc(scrnofs, BytesPerLine-Movebytes);
  end;
end;
}
External; {$L RESTRECT.OBJ}

Procedure SaveRectangle (x,y,width, depth: word; Destination: pointer);
{
  Calls save rect with the right paameters to save the rectangle whose top
  left corner is (x,y) (in global text screen coords) and whose dimensions
  are (width, depth).
}
begin
  if (x >= 1) and (y>=1) and (x+width-1 <= 80) and (y+depth-1 <= 24) then
    SaveRect (Ptr(ScreenSeg,2*((y-1)*80+(x-1))),
              width shl 1,
              80*2,
              depth,
              Destination);
End; {SaveRectangle}

Procedure RestoreRectangle (x,y,width, depth: word; Source: pointer);
{
  Calls RestRect with the right paameters to restore the rectangle whose top
  left corner is (x,y) (in global text screen coords) and whose dimensions
  are (width, depth).
}
begin
  if (x >= 1) and (y>=1) and (x+width-1 <= 80) and (y+depth-1 <= 24) then
    RestRect (Ptr(ScreenSeg,2*((y-1)*80+(x-1))),
              width shl 1,
              80*2,
              depth,
              Source);
End; {SaveRectangle}

Procedure RangeSet(var v: byte; val,Lolimit,Hilimit: byte);
{
  sets v to val, but not greater than HiLimit and not less than LoLimit;
}
begin
  v := val;
  if v > HiLimit then v := HiLimit else if v < LoLimit then v := LoLimit;
End; {RangeSet}

{=====METHODS==}

{=====TState METHODS==}
Procedure TState.SetWindow;
begin
  RangeSet (x1,xl1,1,80);
  RangeSet (y1,yt1,1,24);
  RangeSet (x2,xbr,1,80);
  RangeSet (y2,ybr,1,24);
  window (x1,y1,x2,y2);
End; {TState.SetWindow}

Procedure TState.SetTextAttr;
begin
  RangeSet (FCol, txcol, 0,15);
  RangeSet (BCol, bkcol, 0,15);
  TextColor (FCol);
  TextBackGround(BCol);
End; {TState.SetTextAttr}

```

```

Procedure Tstate.SetCursor;
begin
  RangeSet (cx,x,1,80);
  RangeSet (cy,y,1,24);
  gotoxy(cx,cy);
End; {TState.SetCursor}

{=====TextWin METHODS=====}
Constructor TextWindow.Init;
{}Procedure SetBorder;
{}const
{}  TopLeft  = '*';
{}  BotLeft  = '*';
{}  TopRight = '*';
{}  BotRight = '*';
{}  Horiz    = '*';
{}  Vert     = '*';
{}var
{}  s: string[80];
{}  i:1..80;
{}begin
{}  window (1,1,80,24);
{}
{}  fillchar (s[1],w,Horiz);
{}  s[0] := char(w);
{}  s[1] := BotLeft;
{}  s[w] := BotRight;
{}  gotoxy (x1,y2); write(s);
{}
{}  Insert (Title,s,2);
{}  s[0] := char(w);
{}  s[1] := TopLeft;
{}  s[w] := TopRight;
{}  gotoxy (x1,y1); write(s);
{}
{}  for i := y1+1 to y2-1 do
{}  begin
{}    gotoxy (x1,i); write (Vert);
{}    gotoxy (x2,i); write (Vert);
{}  end;
{}
{}  inc(x1); inc(y1); dec(x2); dec(y2); dec(d,2); dec(w,2);
{}  window (x1,y1,x2,y2);
{}End; {SetBorder}

begin
  RangeSet (x1,x,1,79);      {use 79 instead of 80 otherwise }
  RangeSet (y1,y,1,24);      {writing to 80,24 causes scroll }
  RangeSet (w,width,1,79);
  RangeSet (d,depth,1,24);
  RangeSet (x2,x+w-1,1,79);
  RangeSet (y2,y+d-1,1,24);
  RangeSet (w,w,1,x2-x1+1);
  RangeSet (d,d,1,y2-y1+1);
  if txcol <> bkcol
  then begin
    RangeSet (Fcol,txcol,0,15);
    RangeSet (Bcol,bkcol,0,15);
  end
  else begin
    Fcol := TextState.Fcol;
    Bcol := TextState.Bcol;
  end;

  getmem(OImage,w*d*2);
  SaveRectangle (x1,y1,w,d,OImage);

  Cx := 1;
  Cy := 1;
  Activate;
  clrscr;

  Border := FALSE;
  If bord and (w > 2) and (d > 2) then
  begin
    Border := TRUE;
    SetBorder;
  End; {If}
End; {TextWin.Init}

```

```

Procedure TextWindow.Activate;
begin
  window (x1,y1,x2,y2);
  textcolor (FCol);
  textbackground (BCol);
  gotoxy (Cx, Cy);
  Active := TRUE;
End; {TextWindow.Activate}

```

```

Procedure TextWindow.Deactivate;
begin
  If Active then with TextState do
  begin
    window (x1,y1,x2,y2);
    textcolor (FCol);
    textbackground (BCol);
    gotoxy (CX, CY);
    Active := FALSE;
  End; {If}
End; {TextWindow.Deactivate}

```

```

Destructor TextWindow.Done;
begin
  Deactivate;
  if Border then
  begin
    dec(x1); dec(y1);
    inc(x2); inc(y2);
    inc(d,2); inc(w,2);
  end;
  RestoreRectangle (x1,y1,w,d,OImage);
  freemem(OImage,w*d*2);
  OImage := NIL;
End; {textWindow.Done}

```

```

{=====END OF METHODS==}

```

```

VAR
  driver, mode: integer;
BEGIN
  DetectGraph (driver, mode);
  ScreenSeg := $B800;
  if (driver = HercMono) or (driver = EgaMono) then ScreenSeg := $B000;
  TextState.SetWindow (1,1,80,24);
  TextState.SetTextAttr
    (Mem[ScreenSeg:1] AND $F, (Mem[ScreenSeg:1] AND $F0) shr 4);
  TextState.SetCursor (1,1);
END.

```

B1.19.1 SAVERECT.ASM

```
.MODEL TPASCAL
.CODE
IDEAL
```

```
;Procedure SaveRect (ScreenAddr: pointer;
;                   MoveBytes: word;
;                   BytesPerLine: word;
;                   NoOfLines: word;
;                   Destination: pointer);
;
;Defined in unit TEXTWIN.PAS.
```

```
;This saves parts of lines of an area of memory such as the screen. Each line
;is BytesPerLine long, of which MoveBytes are saved, from the same offset in
;the line as the starting byte. The bytes are saved at Destination. ScreenAddr
;and Destination are assumed to be such that segment boundaries will not be
;exceeded. Moving is done from low to high memory. No range checking is done
;on parameters - these are assumed to be:
;  NoOfLines > 0
;  BytesPerLine > 0
;  0 < MoveBytes <= BytesPerLine
```

```
PROC SaveRect NEAR ScreenAddr: dword, MoveBytes:word, BytesPerLine:word, NoOfLines:word,
Destination:dword
```

```
PUBLIC SaveRect
    cld                                ;count up for block transfers
    mov bx, [BytesPerLine]             ;bx = BytesPerLine-MoveBytes
    sub bx, [MoveBytes]

    lds si, [ScreenAddr]              ;ds = scrnseg := Seg(ScreenAddr^)
                                        ;si = scrnofs := Ofs(ScreenAddr^)
    les di, [Destination]             ;es = destseg := Seg(Destination^)
                                        ;di = destofs := Ofs(Destination^)

    mov cx, [NoOfLines]               ;for lines = 1 to NoOfLines do
LineLoop:                             ;begin
    push cx
    mov cx, [MoveBytes]               ; for bytes := 1 to MoveBytes do
    rep movsb                         ; begin
                                        ;  Mem[destseg: destofs] := Mem[scrnseg:scrnofs]
                                        ;  inc (scrnofs)
                                        ;  inc (destofs)
                                        ; end
    add si, bx                        ; inc(scrnofs, BytesPerLine-Movebytes)
    pop cx
    loop LineLoop                     ;end;

    mov ax, @data
    mov ds, ax
    ret
ENDP SaveRect
END
```

B1.19.2 RESTRECT.ASM

```
.MODEL TPASCAL
.CODE
IDEAL
```

```
;Procedure SaveRect (ScreenAddr: pointer;
;                   MoveBytes: word;
;                   BytesPerLine: word;
;                   NoOfLines: word;
;                   Source: pointer);
;
;Defined in unit TEXTWIN.PAS.
```

```
;This restores parts of lines of an area of memory such as the screen. Each line
;is BytesPerLine long, of which MoveBytes are restored, to the same offset in
;the line as the starting byte. The bytes are restored from Source. ScreenAddr
;and Source are assumed to be such that no segment boundary can be exceeded
;Moving is done from low to high memory. No range checking is done on
;parameters - these are assumed to be:
;  NoOfLines > 0
;  BytesPerLine > 0
;  0 < MoveBytes <= BytesPerLine
```

```
PROC RestRect NEAR ScreenAddr: dword, MoveBytes: word, BytesPerLine: word, NoOfLines:
word, Source: dword
PUBLIC RestRect
    cld                                ;count up for block transfers
    mov bx, [BytesPerLine]             ;bx = BytesPerLine- MoveBytes
    sub bx, [MoveBytes]

    les di, [ScreenAddr]              ;es = scrnseg := Seg(ScreenAddr^)
                                        ;di = scrnoofs := Ofs(ScreenAddr^)
    lds si, [Source]                  ;ds = srceseg := Seg(Source^)
                                        ;si = srceofs := Ofs(Source^)

    mov cx, [NoOfLines]                ;for lines = 1 to NoOfLines do
LineLoop:                             ;begin
    push cx
    mov cx, [MoveBytes]                ; for bytes := 1 to MoveBytes do
    rep movsb                          ; begin
                                        ; Mem[scrnseg:scrnoofs] := Mem[srceseg:srceofs]
                                        ; inc (scrnoofs)
                                        ; inc (srceofs)
                                        ; end
    add di, bx                         ; inc(scrnoofs, BytesPerLine-Movebytes)
    pop cx
    loop LineLoop                     ;end;

    mov ax, @data
    mov ds, ax
    ret
ENDP RestRect
END
```

B1.20 READELEM.PAS

```
PROGRAM ReadElements;
USES BCRTYPES;
{
Utility to read the contents of an Elements (.E) file and
writes them as a text to a .ETX file.

Usage: READELEM <Callname>

Example usage: READELEM redbla
-reads REDBlA.E and outputs to REDBlA.ETX
}

var
  elems:file of elementtype;
  out: text;
  e: elementtype;
  i:classtype;

begin
  assign(elems,paramstr(1)+' .e');
  {$I-} reset(elems); {$I+}
  if IOresult <> 0 then
    begin
      writeln ('Error opening ',paramstr(1),' .E');
      halt;
    end;

  assign (out, paramstr(1)+' .etx');
  rewrite (out);

  writeln ('Reading ',paramstr(1),' .E', ' and writing to ',paramstr(1),' .ETX');
  writeln (out,'Element text file');
  writeln (out);

  while not eof(elems) do
    begin
      read(elems,e);
      writeln (out,'DURN ',e.durn:0:2,
        '   FREQ ',e.freq:0:2,
        '   CLASS ',ClassName[e.class]);
      for i := single to noise do write (out,classname[i],': ',e.occurrence[i]:0:3,' ');
      writeln (out);
      writeln (out);
    end;
  close (elems);
  close (out);
end.
```

B1.21 READLPF.PAS

```
PROGRAM ReadLinkedPeakFrames;
{
  Utility to read the contents of a Linked Peak Frames (.LPF) file and output
  as text to the standard output.

  Run without parameters to see usage.
}
USES BCRTYPES, BCDMANIP;
var
  lpf:file of peakframe;
  out: text;
  pf: peakframe;
  i: word;
  lpfframes, frame, startframe, endframe: longint;
  error: integer;

begin
  if paramcount <> 4 then
  begin
    writeln ('Reads the contents of a linked frames file and writes them');
    writeln ('as text to standard output. Outputs portions of file');
    writeln ('specified in frames. Each frame is separated by a blank line.');
```

writeln;

writeln ('READLPF <Call name> <peak frames suffix> <start frame (first =0)>');

writeln (' <end frame (last = -1)>');

writeln;

writeln('Example:');

writeln(' READLPF redbl a 10 20');

writeln('Reads frames 10 to 20 (inclusive) of redbla.lpf');

halt;

end;

assign(lpf,paramstr(1)+paramstr(2)+'.lpf');

{I-} reset(lpf); {I+}

if IOresult <> 0 then

begin

 writeln ('Error opening ',paramstr(1)+paramstr(2)+'.lpf');

 halt;

end;

assign (out, '');

rewrite (out);

PutCurrentProcess ('CLASSIFY'); {So that we don't have to add to all bcdacces.map files}

BCD('read '+paramstr(1)+paramstr(2)+'.sf Number_of_frames: longint', lpfframes);

val (paramstr(3), startframe, error);

if error <> 0 then startframe := 0;

endframe := 0;

val (paramstr(4), endframe, error);

if (error <> 0) or (endframe = -1) then endframe := lpfframes;

writeln (out,'Linked peak frames file: '+paramstr(1)+paramstr(2)+'.LPF');

writeln (out,'Peaks per frame: ', PeaksPerFrame);

writeln (out,'From frame: ', startframe, ' to frame: ', endframe);

writeln (out, 'Format: <frequency> <magnitude> <link>');

seek (lpf, startframe);

frame := startframe;

repeat

 writeln (out);

 if not eof (lpf) then read (lpf, pf);

 for i := 1 to PeaksPerFrame do with pf[i] do

 writeln (out, freq,' ', mag,' ',link);

 inc (frame);

until (frame > endframe);

close (lpf);

close (out);

end.

B1.22 READSF.PAS

```
PROGRAM ReadSpectralFrames;
{
  Utility to read the contents of a Spectral Frames (.SF) file and output
  to standard output.

  Run without parameters to see usage.
}

USES BCDMANIP;

var
  sf:file of real;
  out: text;
  kHzPerSample, component: real;
  i, fftlength: word;
  sframes, frame, startframe, endframe: longint;
  error: integer;

begin
  if paramcount <> 4 then
    begin
      writeln ('Reads the contents of a spectral frames file and writes them');
      writeln ('as text to standard output. Outputs portions of file');
      writeln ('specified in frames. Each frame is separated by a blank line.');
```

writeln;

```
writeln ('READSF <Call name> <spectral frames suffix> <start frame (first =0)>');
writeln ('          <end frame (last = -1)>');
writeln;
writeln('Example:');
writeln('          READSF redbl a 10 20');
writeln('Reads frames 10 to 20 (inclusive) of redbla.sf');
halt;
end;

assign(sf,paramstr(1)+paramstr(2)+'.sf');
{$I-} reset(sf); {$I+}
if IOresult <> 0 then
  begin
    writeln ('Error opening ',paramstr(1)+paramstr(2)+'.sf');
    halt;
  end;

assign (out, '');
rewrite (out);

PutCurrentProcess ('SPECTRA');
BCD('read '+paramstr(1)+'.ads Frequency_resolution: real', kHzPerSample);
PutCurrentProcess ('PEAKPICK');
BCD('read '+paramstr(1)+paramstr(2)+'.sf FFT Length: word', fftlength);
BCD('read '+paramstr(1)+paramstr(2)+'.sf Number_of_frames: longint', sframes);

val (paramstr(3), startframe, error);
if error <> 0 then startframe := 0;
endframe := 0;
val (paramstr(4), endframe, error);
if (error <> 0) or (endframe = -1) then endframe := sframes;

writeln (out,'Spectral frames file: '+paramstr(1)+paramstr(2)+'.SF');
writeln (out,'Samples per frame: ', fftlength div 2);
writeln (out,'From frame: ', startframe, ' to frame: ', endframe);
writeln (out,'Format: <frequency> <magnitude>');

seek (sf, (fftlength div 2)*startframe);
frame := startframe;

repeat
  writeln (out);
  for i := 1 to fftlength div 2 do
    if not eof(sf) then
      begin
        read(sf,component);
        writeln (out, (i-1)*kHzPerSample, ' ', component);
      end;
    inc (frame);
  until (frame > endframe);

  close (sf);
  close (out);
end.
```

B1.23 UPDATE.PAS

```
{ $M 4096 0 0 }

PROGRAM UPDATE;
{
  Program to update data files - convert them to later formats.
  Checks the date of all files according to the filespec (first parameter)
  and runs an update program on the files if necessary. The map file which
  specifies how to perform updates is passed as the second parameter to this
  program. See UPDATE.XPL, or run without parameters, to get explanation.
  UPDATE.XPL should reside in the same directory as UPDATE.EXE.
}

USES dos, GENLIB;

TYPE
  BCRupdateObj =
    object
      MapExists: boolean; {Tells whether the map file exists}

      UpdateFile: text; {The map file}

      constructor Init (FileName: PathStr);
      function TypeFound (Path: PathStr): boolean;
      procedure CallUpdates (Path: PathStr; FileDate: longint);
      destructor Done;
    end;

Constructor BCRupdateObj.Init;
{
  assigns the Updatefile, checks if it exists (and sets MapExists accordingly)
  and initialises CurrentLine to 0.
}
begin
  assign(UpdateFile, FileName);
  {$I-} reset (UpdateFile); {$I+}
  MapExists := (IOresult = 0)
End;

Function BCRupdateObj.TypeFound;
{
  Searches the update file for a line containing, as the first word after
  'SPEC:', a path spec matching the passed-in path. Returns TRUE if such a line
  was found in the file, else FALSE. Leaves the file position pointing to the
  line after the found line.
}
var
  s: stringseq;
  line: string;
begin
  s.init ('');
  reset(UpdateFile);
  While not (
    eof(UpdateFile) or
    (
      (UpString(s.GetWord) = 'SPEC:') and
      FitsSpec(Path, UpString(s.GetWord))
    )
  ) do
  begin
    readln(UpdateFile, line);
    s.init(line);
  End {While};

  TypeFound := not eof(UpdateFile);
End {TypeFound};

Procedure BCRupdateObj.CallUpdates;
{
  Checks all the "Before:" dates from the current update file position onwards
  until the next EXT: <extension> line or end of file, to see if they are
  earlier than the passed-in date (which is a TP packed DateTime stamp). If so,
  the corresponding update program is executed with the passed-in path as a DOS
  parameter. Note that the program keeps checking the rest of the dates in the
  list against the original passed-in date, so a single file may go through
  each update until it reaches the end of the list. However it only makes sense
  to put dates in ascending order in the update file, unless the corresponding
  update program can update any version.
}
var
```

```

line: string;
s: StringSeq;
datestr: string[10];
error: boolean;
err: integer;
DateAndTime: DateTime;
PackedDate: longint;
PDir, SourceDir: DirStr;
PName: NameStr;
PExt: ExtStr;

begin
  fsplit (paramstr(0),SourceDir, PName, PExt);
  writeln ('Looking for updates to ',Path);
  Repeat
    readln(UpdateFile,line);
    s.Init (line);
    if UpString(s.Getword) = 'BEFORE:'
    then begin
      error := FALSE;
      datestr := s.GetWord;
      fsplit (s.GetWord, PDir, PName, PExt);
      if PDir = '' then PDir := SourceDir;
      if (PExt = '') or (PExt = '.') then PExt := '.EXE';
      if (datestr[3] = '/') then datestr[3] := ' ' else error := TRUE;
      if (datestr[6] = '/') then datestr[6] := ' ' else error := TRUE;
      s.Init (datestr);
      With DateAndTime do
        begin
          Sec := 0;
          Min := 0;
          Hour := 0;
          val (s.GetWord, Day, err);
          if err = 0 then val (s.GetWord, Month, err);
          if err = 0 then val (s.GetWord, Year, err);
          if err = 0 then error := (Year < 1980);
          if err = 0 then packtime (DateAndTime,PackedDate);
        end;
      if (err = 0) and not error
      then if PackedDate > FileDate
        then begin
          writeln ('Updating ',Path,' to ',datestr);
          exec (PDir+PName+PExt,Path);
          if doserror <> 0
          then writeln ('Dos error ',doserror, ' executing ',
            PDir+PName+PExt);
        end
        else
        else runerror(UserMessage('Invalid date format updating '+Path+
          '. Should be DD/MM/YYYY, Year > 1979'));
      end
    else s.Init (line);
  Until (UpString(s.GetWord) = 'SPEC:') or eof(UpdateFile);
End; {CallUpdates}

Destructor BCRupdateObj.Done;
begin
  close (UpdateFile);
End; {Done}

VAR
  DirInfo: SearchRec;
  BCRupdate: BCRupdateObj;
  Dir, SourceDir: DirStr;
  Name: NameStr;
  Ext: ExtStr;

BEGIN
  fsplit (paramstr(0), SourceDir, Name, Ext);
  if ParamCount = 2 then
  begin
    fsplit (paramstr(2),Dir,Name,Ext);
    if Dir = ''
    then BCRupdate.Init (SourceDir+Name+Ext)
    else BCRupdate.Init (ParamStr(2));
    if BCRupdate.MapExists
    then begin
      findfirst(ParamStr(1), Archive, DirInfo);
      fsplit (ParamStr(1),Dir,Name,Ext);
      while DosError = 0 do
        begin
          writeln ('Checking ',DirInfo.Name);
          if BCRupdate.TypeFound (DirInfo.Name)
          then BCRupdate.CallUpdates (Dir+DirInfo.Name,DirInfo.Time);
          findnext(DirInfo);
        end
      end
    end
  end
end

```

```
        end;
    end
    else writeln ('Could not find map file',Paramstr(2));
    end
    else showfile (SourceDir+'UPDATE.XPL');
END.
```

B1.24 PFUPDAT.PAS

```
PROGRAM PF_UPDATE;
{
  For use with the UPDATE utility

  Creates a Linked Peak Frames (.LPF) file from the obsolete Peak Frames (.PF)
  file. The frames of Peak Frames are assumed to be sorted in order of frequency.
}
USES Crt,Dos, BCRTYPES;
TYPE
  oldpeakframe = array [1..PeaksPerFrame] of
    record
      freq, mag: real;
    end;

VAR
  Old_Peak_frames: file of oldpeakframe;
  Linked_Peak_frames: file of peakframe;
  opf: oldpeakframe;
  pf: peakframe;
  i: 1..PeaksPerFrame;
  Dir:DirStr;
  Name:Dos.NameStr;
  Ext:ExtStr;
  ch: char;

BEGIN
  fsplit (ParamStr(1),Dir,Name,Ext);
  assign (Old_Peak_frames,Dir+Name+'.PF');
  {$I-} reset (Old_Peak_frames); {$I+}
  if IOResult = 0
  then begin
    assign(Linked_Peak_frames, Dir+Name+'.LPF');
    {$I-} reset(Linked_Peak_frames); {$I+}
    ch := 'Y';
    If IOResult = 0
    then repeat
      writeln('The linked peak file (LPF.DAT) already exists, '+'
        'overwrite? (Y/N)');
      ch := readkey;
    until upcase(ch) in ['Y','N'];

    If upcase(ch) = 'Y' then
    begin
      rewrite (Linked_Peak_frames);

      While not eof(Old_Peak_frames) do
      begin
        read (Old_Peak_frames, opf);
        fillchar (pf, sizeof(pf), #0);
        for i := 1 to PeaksPerFrame do
        begin
          pf[i].freq := opf[i].freq;
          pf[i].mag := opf[i].mag;
        end;
        write (Linked_Peak_frames, pf);
      End;{While}
      close (Old_Peak_frames);
      close(Linked_Peak_frames);
    End{If}
  end
  else writeln ('Error opening '+Dir+Name+'.PF');
END.
```

B1.25 SSUPDATE.PAS

```
program SS_UPDATE;
{
  For use with UPDATE utility.
  Strips the header off old .SS files to make them work with later versions
  of the BCR system.
}

uses Dos,BCDMANIP;

var
  SS, New_SS: file;
  byt6: array[1..6] of byte;
  byt: byte;
  Dir: DirStr;
  Name:NameStr;
  Ext:ExtStr;

begin
  putcurrentprocess ('SONAGRAM'); {masquerade as SONAGRAM process}

  fsplit (ParamStr(1),Dir,Name,Ext);
  assign (SS,Dir+Name+'.SS');
  {$I-} reset (SS,1); {$I+}
  if IOResult = 0
  then begin
    assign (New_SS,'NEWSS.DAT');
    rewrite (New_SS,1);

    blockread (SS,byt6,6);
    BCD ('write '+ParamStr(1)+' LinesPerMsec: real',real(byt6));
    blockread (SS,byt6,6);
    BCD ('write '+ParamStr(1)+' PixelsPerkHz: real',real(byt6));

    while not eof (SS) do
    begin
      blockread (SS,byt,1);
      blockwrite (New_SS,byt,1);
    end;

    close (SS);
    close(New_SS);

    erase (SS);
    rename (New_SS,Dir+Name+'.SS');
  end
  else writeln('Error opening '+Dir+Name+'.SS');
end.
```

B1.26 CFUPDATE.PAS

```
Program CF_UPDATE;
{
  For use with the UPDATE utility.

  Explains how to update a .CF file from before 17/5/90. In the future it may
  be able to actually call CLASS to do the update automatically - as soon as
  an automated way of reading the data object dependencies is developed.
}

USES dos;
VAR
  Dir:DirStr;
  Name:NameStr;
  Ext:ExtStr;
  Classified_frames: file;
BEGIN
  fsplit (Paramstr(1),Dir,Name,Ext);
  assign(Classified_frames, Dir+Name+'.CF');
  {$I-} reset (Classified_frames); {$I+}
  if IOresult = 0
  then begin
    writeln ('This is a classified frames file made before the magnitude');
    writeln ('Information was being included. To update it, you will need');
    writeln ('to run CLASS. Look in the BCD file for this bird and find');
    writeln ('under which .SF and .PF files ',paramstr(1),' occurs, eg for REDB1,');
    writeln ('REDB1C.CF might be under REDB1B.PF which in turn might be');
    writeln ('under REDB1A.SF - so run CLASS A B C to create the new REDB1C.CF');
  end
  else writeln ('Error opening ',Dir,Name,'.CF');
END.
```

B1.27 SONAPS.PAS

```
PROGRAM SONAPS;
{
  Utility to output the contents of a Sonagram Storage (.SS) file in a similar
  manner to drawer, but to a postscript printer rather than the screen.

  Run without parameters to see usage.
}

USES BCRTYPES, BCDMANIP, GENLIB;

VAR  Sonagram_storage: file of segmenttype;
     PSfile: text;
     LinesPerPage: longint; {Number of horizontal lines per sonagram page.}
     ScreenWidth: longint;
     totallines: longint;
     LinesPerMsec, PointsPerkHz: real;
     CallName, SpecName, SonaName: string[12];

CONST
  XORIGIN = 0;
  YORIGIN = 0; {absolute coords of the sonagram origin}

Procedure DrawAxes (GraphTitle: string; TimeAxisAbs: boolean; MarkTime: word);
var
  unseenmarks: longint;
  tAxis, fAxis, EndtAxis, EndFAxis: word;

Procedure FreqAxis;
var
  axisdrawn, marklabel: real;
  firstdigit: string[1];
begin
  writeln (PSfile, '% FreqAxis');

  (* moveto (tAxis,fAxis);      {draw the axis line}                *)
  writeln (PSfile,fAxis,' ',tAxis,' m');

  (* lineto (EndfAxis,fAxis);   *)
  writeln (PSfile,fAxis,' ',EndfAxis,' lineto');

  (* settextjustify (centertext, bottomtext);                       *)
  {Will use rightjustshow to get the same effect}

  unseenmarks := -1;      {This and the 'if' ensure the zero label is drawn}
  if tAxis <> XORIGIN then unseenmarks := trunc((tAxis-XORIGIN)/PointsPerkHz);
  axisdrawn := (unseenmarks+1)*PointsPerkHz - (tAxis-XORIGIN);

  While axisdrawn <= EndfAxis - tAxis do    {ticks and labels}
  begin
    (* moveto (tAxis,fAxis);      {axis tick}                        *)
    writeln (PSfile,fAxis,' ',tAxis,' m');

    (* moverel (round(axisdrawn),0); *)
    writeln (PSfile,'0 ',round(axisdrawn),' rm');

    (* linerel (0,-4); *)
    writeln (PSfile,'-4 0 adjustxy rl');

    (* moverel (0,-2);      {axis tick label} *)
    writeln (PSfile,'-2 0 adjustxy rm');

    marklabel := (tAxis-XORIGIN+axisdrawn)/PointsPerkHz;

    (* OutInvertText (strung(1,1,marklabel)); *)
    writeln (PSfile,'( ',marklabel:1:1,' ) rightjustshow');

    axisdrawn := axisdrawn+PointsPerkHz;
  End;

  writeln (PSfile,'stroke');

  writeln (PSfile, '% return FreqAxis');
  writeln (PSfile);
End(Procedure FreqAxis);

Procedure TimeAxis;
var
  realdiv, lastmark, axisdrawn: real;
  TimeLines, previousmarks: longint;
```

```

begin
  writeln (PSfile, '% TimeAxis');

  (* moveto (tAxis, fAxis);           {axis line}      *)
  writeln (PSfile, fAxis, ' ', tAxis, ' m');

  (* lineto (tAxis, EndtAxis);        *)
  writeln (PSfile, EndtAxis, ' ', tAxis, ' lineto');

  axisdrawn := 0;
  If TimeAxisAbs then
  begin
    TimeLines := totallines-LINESPERPAGE+fAxis-YORIGIN;
    realdiv := TimeLines / (MarkTime*LinesPerMsec);
    lastmark := Int(realdiv)*(MarkTime*LinesPerMsec);
    if TimeLines-lastmark < 1e-4
    then axisdrawn := 0
    else axisdrawn := lastmark + MarkTime*LinesPerMsec - TimeLines;
  end; {If}

  (* settextrjust (righttext, centertext);             *)
  {will use topjustshow to achieve this affect}

  While axisdrawn <= (EndtAxis-fAxis) do
  begin
    (* moveto (tAxis, fAxis);           {axis tick mark} *)
    writeln (PSfile, fAxis, ' ', tAxis, ' m');

    (* moverel (0, round(axisdrawn));      *)
    writeln (PSfile, round(axisdrawn), ' 0 rm');

    (* linerel (-7,0);                    *)
    writeln (PSfile, '0 -4 adjustxy rl');

    {axis tick label}
    (* moverel (-,0);                    *)
    writeln (PSfile, '0 -2 adjustxy rm');

    if TimeAxisAbs
    then (* OutInvertText (Strung(4,2,(totallines-LINESPERPAGE+fAxis-
      YORIGIN+axisdrawn)/LinesPerMsec/1000)) *)
      writeln (PSfile, '(', (totallines-LINESPERPAGE+fAxis-YORIGIN+axisdrawn)
        /LinesPerMsec/1000:4:2, ') topjustshow')

    else (* OutInvertText (Strung(4,2,axisdrawn/LinesPerMsec/1000)); *)
      writeln (PSfile, '(', axisdrawn/LinesPerMsec/1000:4:2,
        ') topjustshow');

    axisdrawn := axisdrawn+MarkTime*LinesPerMsec;
  End; {While}

  writeln (PSfile, 'stroke');

  writeln (PSfile, '% return TimeAxis');
  writeln (PSfile);
End{Procedure TimeAxis};

```

```

begin
  {Note: Postscript output is to be horizontal rather than vertical as }
  {for the screen, so X and Y coordinates are swopped.}

  writeln (PSfile);
  writeln (PSfile, '%DrawAxes');

  {axes and tick marks should have width independent of scale}
  writeln (PSfile, '0 setlinewidth');

  (* ChooseBox (tAxis, fAxis, EndfAxis, EndtAxis, selected); *)
  {tAxis, fAxis, EndfAxis and EndtAxis are passed in to DrawAxes}
  {for testing purposes:} tAxis := 0;
                        fAxis := 0;
                        EndfAxis := ScreenWidth;
                        EndtAxis := LinesPerPage;

  (* setcolor (white);                                          *)
  writeln (PSfile, '0 setgray');

  (* gettextsettings (textinfo);                                *)
  {no need to save text info}

```

```

(*)  setusercharsize (1,1,5,3); *)
(*)  settextstyle (defaultfont, vertdir,usercharsize); *)
    {we wont change the character shape, just set the font up - use size 8}
    {to match fairly closely the 8x8 letters on the screen}
    writeln (PSfile,'/Courier findfont AxisTextSize scalefont',
            ' adjustfont setfont');

(*)  Graphtitle := Prompt(400,80,'Plot Title?',Graphtitle,
    (EndtAxis-fAxis) div textwidth('0'), escaped); *)
    {Graphtitle is passed in to DrawAxes}

(*)  settextjustify (righttext, centertext); *)
    {will use centrejustshow to get this effect}

(*)  moveto (EndfAxis,(EndtAxis+fAxis) div 2); *)
    writeln (PSfile,(EndtAxis+fAxis) div 2,' ',EndfAxis,' m');

(*)  OutInvertText (Graphtitle); *)
    writeln (PSfile,'(',Graphtitle,') centrejustshow');

    FreqAxis;
    TimeAxis;

(*)  with textinfo do settextstyle (font, direction, charsize); *)
    with textinfo do settextjustify (horiz, vert); *)
    {don't need to restore anything}

    writeln (PSfile,' %return DrawAxes');
    writeln (PSfile);
End{Procedure DrawAxes};

```

```

TYPE
    intervaledge = (Begint,Endint);

```

```

CONST
    Interval: array[Begint..Endint] of
        record
            line: longint;
            name: string[5];
        end
        = ((line: 100000; name: 'START'),
          (line: 100000; name: 'END'));

```

```

    OverPlot: boolean = FALSE;
    overplotstr: string[12] = 'Overplot OFF';

```

```

CONST
    MAXHALFPAGES = 100; {twice maximum number of sonagram pages expected}

```

```

VAR
    SSPosRecord: array[0..MAXHALFPAGES] of record
        pos:word;
        line:longint;
    end;
    {remembers the file positions of the starts of half-page intervals
    in the sonagram, and their corresponding line numbers}
    LastSSpos: 0..MAXHALFPAGES; {points to the last half page position}

```

```

Procedure DrawSonaPage (Startingline: longint);

```

```

var
    lineno:word;
    segment: segmenttype;
begin
    writeln (PSfile, '% DrawSonaPage');

    {skip over lines until the required line is reached}
    totallines := 0;
    while not eof (Sonagram_storage) and (totallines < StartingLine) do
    begin
        repeat
            if not eof (Sonagram_storage) then
                read (Sonagram_storage, segment);
            until (segment.level >= 16);
            inc(totallines);
        end;

        lineno := 0;
        While (lineno < LINESPERPAGE) do
        begin
            If not eof (Sonagram_storage) then
                begin
                    (*)  moveto (XORIGIN,YORIGIN+lineno); *)
                        writeln (PSfile, YORIGIN+lineno, ' ',XORIGIN,' m');

```

```

(draw a line)
Repeat
  If not eof (Sonagram_storage) then
    begin
      read (Sonagram_storage,segment);
      (* setcolor (segment.level mod 16); *)
      {this is incorporated in the call to 'l'}
      if segment.y <= maxint
        then (* lineto (segment.y+XORIGIN,YORIGIN+lineno) *)
              writeln (PSfile,YORIGIN+lineno, ' ',segment.y+XORIGIN, ' ',
                        15-(segment.level mod 16),' l')
            else runerror(UserMessage('Bad sonagram file'));
        End{If};
      Until segment.level >= 16;
    End{If};

    inc (lineno);
    inc (totallines);

End{While};

writeln (PSfile,'% return DrawSonaPage');
writeln (PSfile);
End{Procedure DrawSonaPage};

VAR
  MaxMag: real; {Maximum magnitude of peak in signal- BCDread by main procedure}
  WindowTime: real; {Length of a single frame in milliseconds}
  peaksexist: boolean; {Indicates whether the peak frame file exists}
  classesexist: boolean; {Indicates whether the classified frame file exists}
  elementsexist: boolean; {Indicates whether the elements file exists}

{$F-}

Procedure Prologue;
{
  Send the initilisation commands for the postscript file - subroutine
  definitions, global variables and global transformations, Lotus
  Manuscript header etc.
}
begin
  writeln (PSfile,'% SONAPS Prologue');

  {Define text size used by DrawAxes procedure and the XXjustshow PS routines}
  writeln (PSfile,'%AxisTextSize 12 def');

  {
    Calculate, and store in AdjX and AdjY, the x and y scaling factors
    required to restore a dimension to its meaning before the user space was
    scaled (eg by Lotus Manuscript) at some later stage. Thus any dimension
    scaled by these factors will display in units of points. Note that the
    routine assumes that no rotation of the user space has taken place.
  }
  writeln (PSfile,
    '1 1 matrix currentmatrix dtransform', {current x and y scaling factors}
    ' /cy exch def /cx exch def'); {stored in cx and cy}
  writeln (PSfile,
    '1 1 matrix defaultmatrix dtransform', {original x and y scaling factors}
    ' cy div exch cx div'); {divide these by cx and cy}
  writeln (PSfile, '/AdjX exch def /AdjY exch def');

  {
    Define the 'adjustfont' routine, which scales a font using AdjX and AdjY
    such that the original point size of the font (as specified in a
    'scalefont' call) is reflected in the shown size even if the user space
    has been changed. Usage:

    FONT adjustfont NEWFONT
  }
  writeln (PSfile, '/adjustfont {AdjX AdjY matrix scale makefont} def');

  {
    Define the 'adjustxy' routine which multiplies an (x,y) pair by AdjX and
    AdjY so that x and y specify points even though the user space has been
    scaled. Note that this is useful only when x and y specify a relative
    movement rather than an absolute position, since we have not taken into
    account the translation of the user space from its original setting
    Usage:

    X Y adjustxy NEWX NEWY
  }
  writeln (PSfile, '/adjustxy {AdjY mul exch AdjX mul exch} def');

```

```

{
  Define three routines which show text such that the current point
  coincides with a point on the string other than the bottom left as for
  normal show. Because there is no 'stringheight' facility, it is not
  possible to make this work for an arbitrary font. Instead the height is
  taken from the variable AxisTextSize since it will be used with a font
  scaled with that factor. Furthermore, the font will be adjusted using
  'adjustfont' so that the height here is also adjusted - multiplied
  by AdjY
  Usage for all the routines is:

  STRING xxjustshow -
}

{'rightjustshow': current point is at centre (y) of right (x) end of text}
writeln (PSfile, '/rightjustshow {dup stringwidth pop neg',
        ' AxisTextSize -2 div AdjY mul rmoveto show} def');

{'centrejustshow': current point is at centre (y) of centre (x) of text}
writeln (PSfile, '/centrejustshow {dup stringwidth pop -2 div ',
        ' AxisTextSize -2 div AdjY mul rmoveto show} def');

{'topjustshow': current point is at top (y) of center (x) of text}
writeln (PSfile, '/topjustshow {dup stringwidth pop -2 div ',
        ' AxisTextSize neg AdjY mul rmoveto show} def');

{
  Define the l routine which draws a line in a particular shade to the
  given coordinates from the current point, making the new point current.
  Usage is:

  X Y GRAYSHADE(0 - 15 where 0 is black, 15 white) l -
}
writeln (PSfile, '/l {15 div setgray 2 copy lineto stroke moveto} def');

{Define subroutines to save file space}
writeln (PSfile, '/m {moveto} def');
writeln (PSfile, '/rm {rmoveto} def');
writeln (PSfile, '/rl {rlineto} def');

writeln (PSfile, '% *for standalone* 0.5 0.5 scale');
writeln (PSfile, '% *for standalone* 80 80 translate');
writeln (PSfile, '% return SONAPS Prologue');
writeln (PSfile);
End; {Procedure Prologue}

VAR
  WindowLength: word;
  SamplingFreq: real;
  FrequencyScale: real; {kHz fitting onto a screen, whatever a screen's width was defined
to be in Sonagram}
  StartTime, EndTime: real; {defines part of graph to be plotted}
  StartLine, EndLine: longint; {as above but in sonagram lines rather than milliseconds}
  error: integer;
  timepertick: real;
  title: string;
  i: byte;

BEGIN
  PutCurrentProcess('SONAPS');

  if paramcount < 3 then
    begin
      writeln ('SONAPS <call name> <spectral frames suffix> <sonagram storage suffix>');
      writeln ('<start time (ms)> <end time (ms)> ["title"] [time per tick]');
      writeln ('e.g:');
      writeln ('          SONAPS redbl a c 0 1000 "test" 50');
      writeln ('sends sonagram REDB1C.SS to PostScript file REDB1C.PS');
      halt(0);
    end;
  CallName := paramstr(1);
  SpecName := paramstr(2);
  SonaName := paramstr(3);

  i := 6; title := '';
  if (paramcount >= i) then title := paramstr(i);
  if (title[1] = '"') then
    begin
      title := copy (title, 2, length(title));
      repeat
        inc(i);
        if (paramcount >= i) then title := title+' '+paramstr(i);
      until (i > paramcount) or (title[length(title)] = '"');
    end;

```

```

if title[length(title)] = ''
then begin
    title := copy(title,1,length(title)-1);
    inc(i);
end
else title := '';
end
else title := '';

if paramcount = 1
then begin
    val (paramstr(i),timepertick,error);
    if error <> 0 then
        runerror(UserMessage('Error in time per tick parameter: '+ paramstr(i)));
    end
else timepertick := 100;

assign (Sonagram_Storage, CallName+SonaName+'.SS');
if FileQuery (Sonagram_Storage,DontAsk) = Exists
then reset (Sonagram_storage)
else runerror(UserMessage('SONAPS could''nt find the '+CallName+SonaName+'.SS file'));

assign (PSfile, CallName+SonaName+'.PS');
rewrite (PSfile);

BCD('read '+CallName+'.ADS '+'Sampling_Frequency: real', SamplingFreq);
BCD('read '+CallName+SpecName+'.SF '+'Window_length: word', WindowLength);
BCD('read '+CallName+SpecName+'.SF '+'CallName+SonaName+'.SS '+'PixelsPerkHz: real',
    PointsPerkHz);
BCD('read '+CallName+SpecName+'.SF '+'CallName+SonaName+'.SS '+'LinesPerMsec: real',
    LinesPerMsec);
BCD('read '+CallName+SpecName+'.SF '+'Max_magnitude: real', MaxMag);
BCD('read '+CallName+SonaName+'.SS '+'Frequency_scale: real', FrequencyScale);

WindowTime := WindowLength/SamplingFreq;
val (paramstr(4), StartTime, error);
val (paramstr(5), EndTime, error);
if error <> 0 then
    runerror(UserMessage('Error in start or end parameters: '+paramstr(4)+' '+paramstr(5)));

StartLine := round(LinesPerMsec*StartTime);
EndLine := round(LinesPerMsec*EndTime);
LinesPerPage := EndLine-StartLine; {Pretend the screen is just big enough}
ScreenWidth := round(FrequencyScale*PointsPerkHz);

totallines := EndLine; {Pretend sonagram has been drawn}
{normally: totallines := 0, then DrawSonaPage must be called before
DrawAxes, so that totallines will be correctly calculated}

LastSSPos := 0;
fillchar (SSPosRecord,sizeof(SSPosRecord),#0);
SSPosRecord[0].line := 0;
SSPosRecord[0].pos := filepos(Sonagram_storage);

{set up header for .EPS format - specify lower left and }
{upper right corner coords in points}
writeln (PSfile,'%!PS-Adobe');
writeln (PSfile,'%BoundingBox: -40 -40 ',LinesPerPage+40,' ',ScreenWidth+40);

Prologue; {Set up postscript procedures, global variables and}
{transformations}

DrawSonaPage (StartLine);
DrawAxes (title,true,round(timepertick));

close (Sonagram_storage);
close (PSfile);
END.

```