EISCAT

Digital Signal Processing: Channel Boards

www.eiscat.se

Address: EISCAT AB
Bengt Hultqvists väg 1
SE-981 92 Kiruna
Sweden
Phone: +46 980 791 50

E-mail: contact.us@eiscat.se

Digital Signal Processing: Channel Boards

The 15 Msamples/s data stream output from the A/D converter is a discrete-time, digital representation ofall information in the 2ndi.f. passband (see The Bandpass Sampling Theorem). It is theoretically possible, and some would probably find it aestethically pleasing, to terminate the receiver hardware at this point, feed the raw data stream into a powerful computer and do all the rest of the signal processing by software.However, after looking at this alternative and some others, the EISCAT design team eventually selected a mixed hardware/software system. In the new receiver, the A/D data stream is first converted to baseband, low-pass filtered, decimated, gated and buffered by one or several *channel boards* before being sent to the process computers. A receiver crate can accommodate six channel boards, so a fully configured receiver can select and process up to six spectral windows simultaneously.

1.1. Functional description

A single channel board replaces nearly all of the analog hardware associated with a receiver channel in the old KST receiver system (2nd local oscillator, 2nd mixer, quadrature detector, post-detection filters), plus the ADC channel ON/OFF feature and the correlator buffer memory.To accomplish all this on a single 6U VME board obviously requires a great deal of complex logic, and so every channel board comprises, among other things,

a data source selector,

a frequency-agile numerically controlled oscillator (NCO),

a complex digital mixer, performing real-to-baseband conversion,

a pair of programmable FIR filter chips, including a decimation feature,

logic to range gate the decimated sample stream,

a ping-pong RAM memory to buffer the gated data stream, and

support logic.

A HSP 45116 ASIC (for full data please refer to http://www.intersil.com/data/fn/fn2/fn2485/ and references therein) contains the NCO and real-to-baseband quadrature down-converter functions. As shown in Figure x, the HSP 45116 is divided into three main sections. The Phase/Frequency Control Section (PFCS) and the Sine/Cosine Section together form a programmable Numerically Controlled Oscillator (NCO). The Complex Multiplier andACcumulator (CMAC) multiplies the output of the NCO with an external data vector.The NCO frequency is programmed by loading a 32 bit frequency word into the PFCS. On the channel boards, the HSP 45116 is clocked at 15.000 MHz by the same clock

signal that drives the A/D conversion system. To generate a frequency f0, thefrequency word should then be set to

(f0 / 15.000) x 232 , where f0 is in MHz.

The PFCS will automatically generate a corresponding time sequence of phase values. The Sine/Cosine Section takes the argument sequence and generates acomplex (cos 2 ? f0 t, sin 2 ? f0 t) sinusoid, which becomes one of the CMAC inputs. The CMAC multiplies this (cos, sin) vector by an external data vector and outputs the result as a complex result vector.

Channel boards can be set under computer control to process either of two data streams, generated by the two A/D converter units on the Pentek 6420 board, running in parallel. The desired data stream is fed in parallel to the real and imaginary ports of the CMAC Vector Input, one sample per clock cycle, and multipliedby the rotating vector from the Sine/Cosine Section. The NCO frequency, f0, is freely selectable up to 15.000 MHz, but should be somewhere in the (8.3 – 14.7) MHz 2nd i.f. range to make sense. It can be set either from the EROS III command line or loaded from a radar-controller driven frequency register stack containing 16 registers (NCO 0.....NCO 15).

The resulting complex Vector Output has one component centred on 2*f0 and another centred on zero frequency. Only the zero frequency component is required in the further processing, so the Vector Output data stream is sent to two paralleled HSP 43220 Decimating Digital Filter chips which perform low pass filtering and decimation (Figure y). The output of the HSP 43220's is a true low sample rate, complex baseband signal which is range gated by the sample gate logic and stored in one page of the dual-page,256 K samples deep buffer memory. At the same time, data stored in the other page is accessible for readout by the crate computer.

1.2. Data source selection

The channel board subsection of the system VME rack can accommodate a maximum of six channel boards. The two 15 MHz data streams from the Pentek 6420 dual channel A/D converter are broadcast to all boards.The boards are arranged in two groups of three. The "left" or L group comprises boards 1 to 3 and the "right"or R group boards 4 to 6. For each group separately, one can select which of the two data sources shall be the active one. This is done in the .tlan file by entering some combination of the following commands at the beginning of each cycle:

```
AD1L       % Data from AD1 to boards 1 – 3
AD1R       % Data from AD1 to boards 4 – 6
AD2L       % Data from AD2 to boards 1 – 3
AD2R       % Data from AD2 to boards 4 – 6
```

Important note: These commands are optional. If not found in your .tlan file, the system defaults to sending AD1 data to all six channel boards (as if you had actually used the command sequence AD1R AD1L).

One obvious application of the selection feature is in dual-beam VHF experiments, which require the whole receiver to be essentially "split in two" to process signals from the two beams independently (cf. CP4BV).

1.3. Loading the NCO frequency table registers

Every channel board should have its frequency register stack loaded before an experiment is started. This is done either from the EROS III command line, or from the **.elan** file, using the command **loadfrequency <freqfile> ch<chno>**

If **<freqfile>** is found and its contents are acceptable, the NCO frequency table register stack on the **ch<chno>** board is loaded with the specified frequencies. Since the boards are initialised one at a time, a separate **loadfrequency** command must be issued for every channel board.

The **<freqfile>** file names must be of the type

$XDIR/ch<chno>_<expname>.nco

Example:
set NCO1 $XDIR/ch1_cp1.nco
loadfrequency $NCO2 ch2

The **<freqfile>** files themselves are plain ASCII text files, which typically look like this (the example is from CP4, channel 1):

```
NCOPAR_VS 0.1
%====================================
%cp4 freq settings
%LO1 298 MHz LO2 84 MHz
%====================================

NCO 0  0
NCO 1  9.8  % f7
NCO 2  9.6  % f6
NCO 3 10.2  % f9
NCO 4 10.0  % f8
```

The file header always looks the same: `NCOPAR_VS 0.1`

Lines beginning with a % are comments only (as in EROS II)
The file body can contain up to sixteen lines. Each line starts with the keyword NCO, followed by

a register address from 0 to 15 and afrequency value in MHz. Comments are allowed after the three mandatory fields if preceeded by a %.

Only those registers which will be used later need to be set; in the extreme case (an experiment requiring only one set of frequencies), the file body could be just one line.

1.4. Setting the NCO at run-time

To select a pre-loaded frequency from the NCO frequency table at run-time, use the following command in the .TLAN file:AT <fsettime> NCOSEL<freqno>

Example: AT 2345 NCOSEL3

The NCO will start to output the selected frequency at approx. <fsettime> + 300 ns.

Important Note: The NCOSEL command is *global*. When issued, it goes to all channel boards in parallel, causing all NCOs to change frequencies at the same time.

NCOSEL is implemented this way mainly for convenience. Most standard experiments *do change all frequencies cyclically* in successive radar cycles, and so it is practical to have a single command per cycle that changes them all at once.

However, if your experiment is of the kind that requires one or several NCO frequencies to be kept while some others are changed, this can be handled just as easily. How to do it is best explained by inspecting the following table:

NCOSEL	f (CH1)	f (CH2)	f (CH3)	f (CH4)	f (CH5)	f (CH6)
0	8.000	8.500	9.000	9.500	10.000	10.500
1	*10.000*	8.500	9.000	9.500	10.000	*11.000*
2	*11.000*	8.500	9.000	9.500	10.000	*12.000*

In this example, only CH1 and CH6 will be required to change frequencies at some point, while the settings of CH2.....CH5 should remain constant. If the same frequency value is entered on every line in the CH2.....CH5 .nco files, e.g. like so:

```
NCOPAR_VS 0.1
%====================================
%demoexp freq settings for CH 2:
%LO1 812 MHz LO2 128 MHz
%====================================
NCO 0 8.500       % almost at band edge !
NCO 1 8.500
NCO 2 8.500
```

the frequency settings of CH2.....CH5 will not change when the NCOSEL command is issued (the effect is that ofre-loading with the same frequency) !Important Note: Use caution if you plan to use this feature in an experiment which requires pulse-to-pulse phase coherence of the whole system. Preliminary tests indicate that phase coherency is generally preserved when a channel is commanded to the same frequency it is already running on, but we cannot yet guarantee that this holds under all circumstances. More information on this will follow later.

1.5. Setting the NCO frequency from the command line

To set the NCO of channel board <chno> to frequency <ncofreq> from the EROS III command line, typesetfrequency <chno><ncofreq>

where <ncofreq> is in MHz. This command can also be used in an .elan file. Instead of specifying a single chno, one can also supply a chnolist. All listed channels will be set to the same ncofreq. This command pokes the <ncofreq> value straight into the NCO chip(s). There is no copy of the value kept in any on-board register.

Important Note: If the radar controller is running a file containing even a single NCOSEL command when the setfrequency command is given, or if such a file is started later, the first execution of the NCOSEL will overwrite the poked frequency setting with some value fetched from the frequency register stack, and the <ncofreq> value will be lost. Thus setfrequency is only useful in cases where no frequency agility at all is required and the radar controller file contains no NCOSEL commands, or for test purposes.

1.6. Table 1: NCO frequency settings for UHF ion line work

Transmit channel number	UHF NCO frequency
F0	14.000
F1	13.700
F2	13.400
F3	13.100
F4	12.800
F5	12.500
F6	12.200
F7	11.900
F8	11.600
F9	11.300
F10	11.000
F11	10.700
F12	10.400
F13	10.100
F14	9.800
F15	9.500

1.7. FIR filter

The HSP 43220 Decimating Digital Filter chips making up the low pass filter part of the channel board are highly complex devices. For the purpose of loading and executing pre-defined filter functions, it is however not necessary to know anything about their inner workings. Interested users who would like to design their own filters are referred to the manufacturer's full technical documentation of the 43220, available on the Web at http://www.intersil.com/data/fn/fn2/fn2486/ and further links therein.

1.8. Sampling interval, decimation factor and how they are related

Important Note: In the old system, the time interval between successive samples on a given channel, tS, was set explicitly by an EROS command. This is no longer so.*In the new system, all FIR filters output data continuously. The data rate out of a specific FIR filter,* dataout_rate, *is some integer fraction of the 15 MHz input rate, defined by the decimation factor,* DF:

dataout_rate = (15 / DF) MHz

The time interval between successive samples, tS, *becomes:*

tS = (DF / 15)μs

*Example:*DF = 15 =>tS = 1 μs
DF = 225 =>tS = 15 μs

The decimation factor must be loaded into the FIR chips at setup time, together with the FIR parameters. **Thus, when you select a filter, you also select a sample rate !**

1.9. Loading the FIR filters; filter library

Every channel board should have its FIR chips loaded before an experiment is started. For convenience, a filter library, located in the directory**/kst/dsp/fir/** has been established at all sites. It contains parameter files defining a number of different Gaussian FIR filters. There are also Postscript graphics files depicting the filter responses, to aid users in selecting the best filter for their application.

The file names are self-explanatory:

b<*bw*>d<*df*>.fir*bw* is the one-sided -3 dB bandwidth in KHz and ***df*** is the decimation factor. Please keep in mind that, since two filter chips are operating in parallel, the full instantaneous bandwidth at the filter output is always twice the one-sided bandwidth. All filters available at present, including all those used in the various Common Programmes, are listed in Table 2.

To load a filter file, use the following command either from the EROS III command line or in your **.elan** file: **loadfilter <chno><FIR_filename>**

The boards are initialised one at a time, so you should issue a separate command for each board. As the FIR loading is a relatively slow process (a large number of parameters must be transferred serially into the FIR chips), re-loading "on the fly" under RC control is strongly discouraged.

Table 2: FIR filter files available in /kst/dsp/fir/

Filename	-3 dB BW (kHz)	Decimation factor	Sample interval (µs)	Used in
b15d225.fir	15	225	15	CP4BVLP/PP
b16d450.fir	16	450	30	
b20d225.fir	20	225	15	
b21d360.fir	21	360	24	
b25d105.fir	25	105	7	
b25d150.fir	25	150	10	CP1LTLP
b25d315.fir	25	315	21	
b30d225.fir	30	225	15	CP7VLP/PP
b35d150.fir	35	150	10	
b42d180.fir	42	180	12	
b75d105.fir	75	105	7	CP1LTA/C
b250d30.fir	250	30	2	

1.10. Sample gate

The data path from the FIR filter to the buffer memory passes through a sample gate. By opening and closing the gate at precisely defined times, the user selects data from specific range intervals for further processing. Thus the sample gate is a functional replacement for the turning on and turning off of individual ADC channels in the old receiver.The sample gate command syntax is almost the same as the “channel” commands in the old system.

To turn on a channel at time <starttime>:

AT <starttime> CH<chno>

To turn off a channel at time <endtime>:

AT <endtime> CH<chno>OFF

Example:
AT 3456 CH3
AT 4567 CH3OFF

Important Note: In the old system, for obscure reasons the CH<chno>OFF command had to be issued prematurely, i.e. <endtime> had to be set to a value about halfway into the last sample period to get the desired number of samples. If issued later, the ADC produced one sample too many.

In the new system, <endtime> is always set to the exact end-time of the desired sampling interval. (<endtime> – <starttime>) must be an integer multiple of the time interval between samples in the FIR output data stream, tS = (DF / 15), otherwise the last sample may be lost.

1.11. Buffer memory

Each channel board has a dual-page, 256 K samples deep, “swinging buffer” memory area. Data from the FIR filter is written into one memory page whenever the sample gate is opened. At the same time, data stored in the other page can be accessed by the Force CPU-50 crate computer.At the end of a data collection period (which can be one or many radar cycles), the radar controller must issue a BUFLIP command:

AT <bufliptime> BUFLIP

This exchanges the roles of the two pages (“flips” them), just as in the old correlator, making the accumulated data accessible for read-out. To off-load the data, the radar controller must then issue a STC (“start compute”) command, which generates an interrupt to the CPU-50:

AT <stctime> STC

The interrupt handler running in the CPU-50responds by launching a real-time software process,/kst/bin/ lag_wrap, which reads data from the board via the VME bus and processes it as required.

Important Note: In the old system, for illogical reasons one had to issue the STC command while at least one channel was still sampling.

In the new system, BUFLIP should only be issued after all channels are closed, but always at least 1 µs before STC, which is the last executable command in a radar cycle. BUFLIP and STC must both be issued no earlier than 15 us before the REP that defines the end of the radar cycle:

<reptime> – 15 us<=<bufliptime><=<stctime><=<reptime>

Example:
AT 1234 ALLOFF
AT 1235 BUFLIP
AT 1236 STC
AT 1250 REP

If this is not followed, data may become lost or corrupted !

1.12. Digital Signal Processing: The “DSP Engine”

1.12.1 /kst/bin/lag_wrap and decodump

/kst/bin/lag_wrap and decodump are two real-time processes, which together form the new receiver system “DSP engine”. They perform the basic signal processing primitives required in an incoherent scatter radar receiver, and so can be regarded as a “software correlator”./kst/bin/lag_wrap runs on the CPU-50 VME Sparc computer, while decodump runs in the experiment server (t45001 in Tromsø, s2501 in Sodankylä or k2501 in Kiruna).

Important Note: /kst/bin/lag_wrap can only run in the CPU50 VME crate computer, since it references a VME back-plane driver library which is used to communicate with the channel boards.

The two routines handle different parts of the processing, as follows:

/kst/bin/lag_wrap reads samples from the channel boards, runs FIR filtering on the sample vector if required and performs all MAC (multiply-and-accumulate) operations intrinsic to forming correlation estimates. It accumulates data for one pre-integration cycle at a time. At the end of the cycle, /kst/bin/lag_wrap writes the accumulated data to a temporary disk file which physically resides on the experiment server disk. There is a separate 100 Mbit/s Ethernet line between the

two machines dedicated to this transfer, allowing /kst/bin/lag_wrap to off-load data at rates of up to 6 – 7 MB/sec, sustained.

Now decodump wakes up in the server. It reads the data off the temporary file, discards certain irrelevant and/or meaningless areas, decodes those results that require decoding (e.g. alternating codes data), formats the results, adds auxiliary data and writes the completed data record to a result file, where the user and/or GUISDAP can retrieve it.

Maybe all this looks as prohibitive as programming the old correlator? Relax – from the user's point of view,/kst/bin/lag_wrap and decodump can be visualised as one process which can read channel board memories and do something useful with the data it finds there. You will need to tell them what to do by creating a few simple files.

1.13. How to use /kst/bin/lag_wrap

At the moment, /kst/bin/lag_wrap can transfer raw data and/or compute

lag profiles,

gated power profiles and

total power estimates.

FIR pre-filtering of the amplitude domain data (e.g. Barker decoding) can be applied before the main processing.

When your experiment is run from the default experiment user (at present /kstdev/), /kst/bin/lag_wrap is already in your path and you don't need to do anything extra to retrieve it.The following three steps are required to get a new experiment going:

Create the required files. As a minimum, you need a set-up file with the name <exp_name>.fil, where <exp_name> is the name of your experiment. Depending on what you want to do, you may also need one or more code files and/or one or more FIR filter coefficient files.

Run /kst/bin/lag_wrap in compile/check mode. Switches -c and -f must be set and<exp_name>.fil specified after -f. Correct any errors. Once your files compile correctly, a file with the name <exp_name>.DECO will be automatically generated. The contents of this file tell decodump what to do.

Start EROS III and start your <exp_name> .elan file. Somewhere in this file you will have a start_recording command; when this is executed, it starts /kst/bin/lag_wrap and decodump in run mode and the data begins to flow! At this time, there will be a dump_map file generated, showing you the layout of the auto-generated result memory area.

Once these three steps have been successfully completed, you don't need to repeat steps 1 and 2 again when starting the experiment – you can just call up your .elan file and go!

1.14. Set-up file syntax and structure

The contents of the set-up file, <exp_name>.fil, is basically a sequence of statements that assignvaluesto control parameters, specifying what /kst/bin/lag_wrap and decodump will do with your data.We have made a deliberate effort to keep the number of control parameters as small as possible without restricting the degrees of freedom too much. Parameter definitions and syntax have been chosen to be self-explanatory wherever possible. Please note that some parameters are mandatory.Several others can only be used with specific type= values and some are relevant only in certain combinations.

While not strictly necessary from the functional point of view, we added two delimiter statements, end_channel and end_type, to the syntax. With the aid of these statements, the /kst/bin/lag_wrap compiler will enforce that the set-up files are always structured by channels and data types in hierarchical order.

The following statements are mandatory:

nr_stc= <number_of_STCs_per_R/C_loop>;

channel= <physical_channel_number>;
end_channel;

type= <type_of_computation>;
end_type;

nr_stc= <number_of_STCs_per_R/C_loop>;

tells the system how many STC commands to expect per integration period. During experiment initialisation, the system reads the radar controller, finds out how many loops per integration have been programmed into it, and multiplies this number by number_of_STCs_per_R/C_loop to find the expected number of STCs per pre-integration. At runtime, this product is compared to the number of STC interrupts actually processed. If the two don't agree, an error message is output on the CPU50 sys-log file. It is normal to see this message appear in the very first integration period, but it should not appear again as long as everything runs OK.

channel= <physical_channel_number>;

and

end_channel;

must always appear as a pair. channel=specifies a physical channel number in the range 1.....6. Everything between this statement and the next end_channel statement will apply only to the specified physical_channel_number.

Between a channel= and a end_channel, there must be inserted one or several block(s) of statements, each bracketed by a

type= <type_of_computation>;

.

end_type;

pair. Every such statement block uniquely specifies a particular type of processing to be applied to some or all of the data found in physical_channel_number.type_of_computation must have a value in the range 0....3:

Raw Data. Data is read from the channel boards, buffered with no processing and transferred to decodump at the end of each pre-integration. This is what you should specify if you want to record raw data for off-line processing.

Lag Profiles. Data read from the channel board is processed into a set of lag profiles, starting with lag-0 and ending with lag-max_lag. All possible cross products are computed. The output vector is the entire set of lag profiles in order of increasing lag index. All lag >0 profiles are padded with zeros at the end, such that all “profiles” are of equal length during the transfer to the server. This is the “standard” computation type which does essentially everything needed for routine operation. For ungated power profile, use type_of_computation= 1with max_lag set to zero.

Gated Power Profile. Basically the same as a lag-0 profile with the results from several samples summed into each output vector location. Included to maintain backward compatibility.

Total Power. Generates a single (or a very few) output value(s) equal to the sum of squares of every point in the vec_len, data_start input vector.

Example: A set-up file for an experiment using channel 1 for raw data taking and channel 4 for lag profiles and gated power profiles could have a structure like this:

```
nr_stc= 34;
channel= 1;
    type=  0;
        .
    end_type;
end_channel;
channel= 4;
    type=  1;
        .
```

```
    end_type;
    type=  2;
        .
    end_type;
end_channel;
```

The full dots indicate where the statement blocks specific to each type should appear.Statements which must appear in every type=block:

vec_len= <number_of_input_samples>;
data_start= <start_address_in_buffer_memory>;

These two statements specify how many samples should be read from the physical_channel_number buffer memory, and where the reading should start from. Please note that the buffer memory addresses start from 0 !

Statements which may appear in any type=block:

fir_len= <number_of_taps>;
fir_file= <firfile>;

When present, these statements will cause the samples to be FIR filtered (e.g. for decoding Barker coded data) before the processing specified by the type= statement takes place. They must appear together; fir_len= is used to specify the number of taps in the FIR filter and fir_file= is used to specify the name of a file specifying the tap weights. One tap per line!

If no FIR filtering is required,simply omit these instructions.

res_mult= <number_of_consecutive_result_vectors>;

This statement requires a bit of explanation. At runtime, /kst/bin/lag_wrap automatically reserves the amount of result memory required to store the results from a single instance of all computations specified by the set-up file, that is, the results produced after a single STC interrupt. At the beginning of every pre-integration, this reserved memory will be cleared and results will then be accumulated in-place for the entire pre-integration cycle.

However, if a res_mult=statement is found in the set-up file, /kst/bin/lag_wrapinstead reserves

(number_of_consecutive_result_vectors) X (the computed number of memory locations)

The effect is that of creating number_of_consecutive_result_vectors consecutive output vectors in the result memory. /kst/bin/lag_wrap will now store data computed after STC number 1 into the first vector, data computed after STC number 2 into the second vector, and so on until all number_of_consecutive_result_vectors have been written to once. It will then start over with the first result vector, but this time it will accumulate rather than overwrite the old data.

As the experienced EISCAT user will see, this can be useful in a number of applications.

Example: When processing alternating codes data, results from different codes must be kept separated until decodump takes over and decodes the whole set. This can be arranged by setting

res_mult= <number_of_STCs_per_code_set>;

Example: If the user wants to record Raw Data (type 0), she sets

res_mult= <number_of_STCs_per_preintegration>;

The effect is that every single raw data sample is buffered individually and never overwritten before being transferred to the server and decodump.

Important Note: In most cases (but not always), the total number of STCs in a pre-integration will be an integer multiple of the res_mult= value for the data to make sense. There is no software check of this at runtime, so use caution !

Type specific statements:

type= 0 (Raw Data) block:

None. However, for the data to make sense, you would normally include a

res_mult= <number_of_STCs_per_preintegration>;

command in every type= 0 block (see above!)

type= 1 (Lag Profile) block:

max_lag= <highest_lag_index_computed>;

specifies the highest lag index to be computed. If omitted, defaults to 0.

code_len= <number_of_bauds_per_code>;

Meaningful only for alternating codes data. Specifies the number of bauds per code sequence (i.e. per IPP).

ac_file= <codefile.txt>;

specifies the name of a code file containing the definition of the alternating codes set used.This is used by the decoding function in decodump. The format of this file is extremely simple: Each row specifies one code. As many rows as there are codes in the set. Positive and negative bauds are specified as 1 and -1 respectively.
n_frac= <fractionality>;

tells decodump to what fractionality the input data has been sampled. This information is required to enable correct decoding of the resulting fractional lags.

Definition: fractionality = (transmitted baud length/sample interval)

fractionality must be an integer. It is the user's responsibility to select baud lengths and sample intervals such that this is met.
sub_int= <number_of_subintegrated_cycles>;

specifies the number of sequential R/C cycles to be accumulated in the same output vector. This can be used when consecutive cycles are logically equivalent (i.e. modulated with the same code, although perhaps not transmitted at the same frequency). Only meaningful when used together with res_mult; if omitted, number_of_subintegrated_cycles defaults to 1.
do_zlag= <zerolagflag>;

decodump normally discards the lag-0 profile data from an alternating codes set, but by setting

do_zlag= 1;

you can force decodump to output also this part of the result vector. Useful for checking purposes.
type= 2 (Gated Power Profile) block:

gating= <gating_value>;

specifies how many samples will be used to compute each point in the output vector.
type= 3 (Total Power) block:

sub_div= <sub_division_value>;

specifies how many output points will be generated by this instance of type= 3. The number_of_input_samples in the input vector will be sub-divided into sub_division_value consecutive pieces and the total power of each piece computed and stored separately. Note that for this to work correctly, number_of_input_samples must be an integer multiple of sub_division_value !
Example: The set-up file for the new CP1-L-T looks like this:

```
% cp1lt.fil
%
% Channel one

channel =1;
%ch_mem_base=387;

% 40 us (F region) power profile, signal
    type=1;
        max_lag=0;
        vec_len=240;
        data_start=0;
    end_type

% Power profile, background
```

```
    type=1;
        max_lag=0;
        vec_len=120;
        data_start=240;
    end_type

% Power profile, calibration
    type=1;
        max_lag=0;
        vec_len=27;
        data_start=360;
    end_type
end_chan

% Channel two

channel =2;
%ch_mem_base=644;

% 350 us long pulse, signal
    type=1;
        max_lag=24;
        vec_len=416;
        data_start=0;
    end_type

% Long pulse, background
    type=1;
        max_lag=24;
        vec_len=202;
        data_start=416;
    end_type

% Long pulse, calibration
    type=1;
        max_lag=0;
        vec_len=26;
        data_start=618;
    end_type

end_chan

% Channel three

channel =3;
%ch_mem_base=594;

% E region power profile from 21 us pulse, sampled at 7 us
    type=1;
        max_lag=0;
        vec_len=309;
        data_start=0;
    end_type

% 16 x 21 us alternating code, sampled at 7 us
    type=1;
        res_mult=32;
        code_len=16;
        n_frac=3;
```

```
        ac_file=ac.txt;
        sub_int=2;
        max_lag=44;
        vec_len=285;
        data_start=309;
    end_type

end_chan

% Channel four

channel =4;
%ch_mem_base=624;

% second E region power profile, signal
    type=1;
        max_lag=0;
        vec_len=309;
        data_start=0;
    end_type

% E region power profile, background
    type=1;
        max_lag=0;
        vec_len=276;
        data_start=309;
    end_type

% E region power profile, background/calibration
    type=1;
        max_lag=0;
        vec_len=39;
        data_start=585;
    end_type
end_chan
```