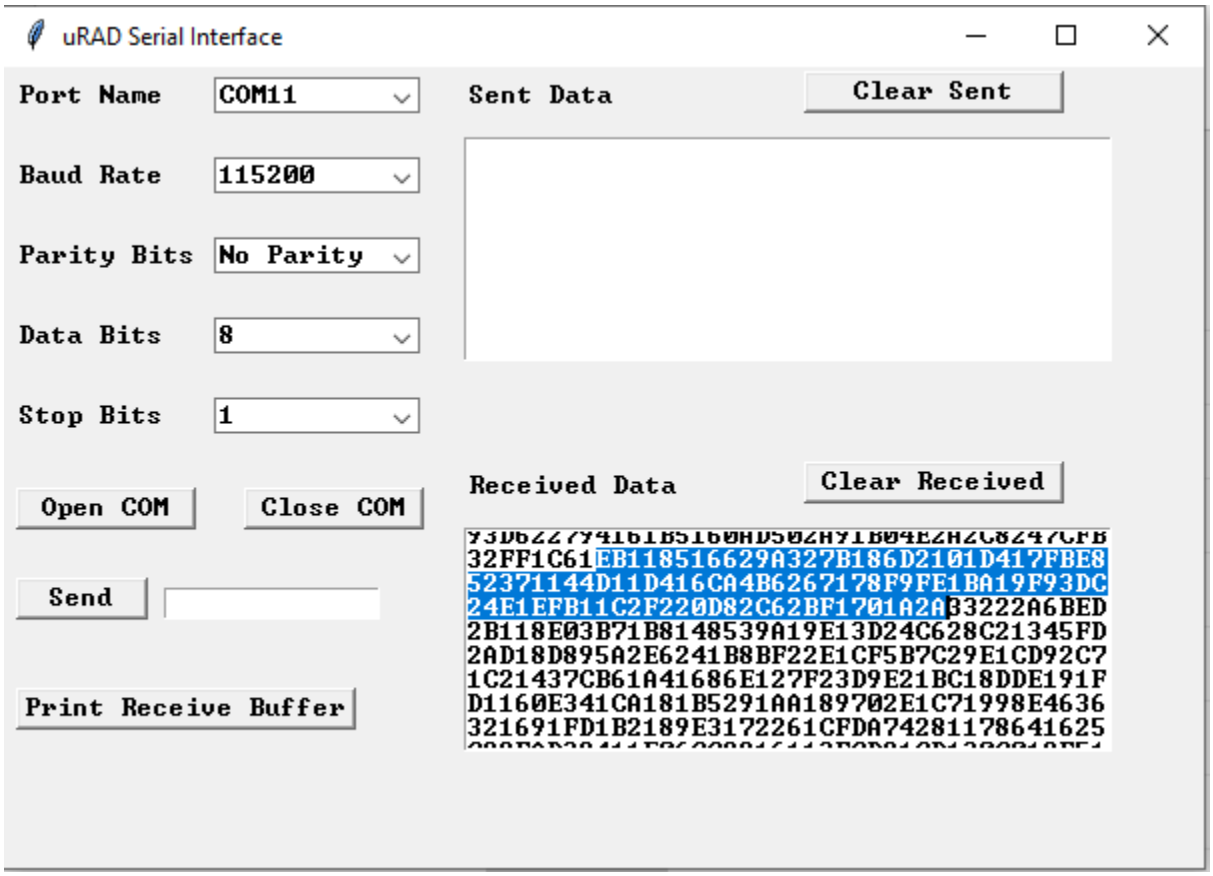
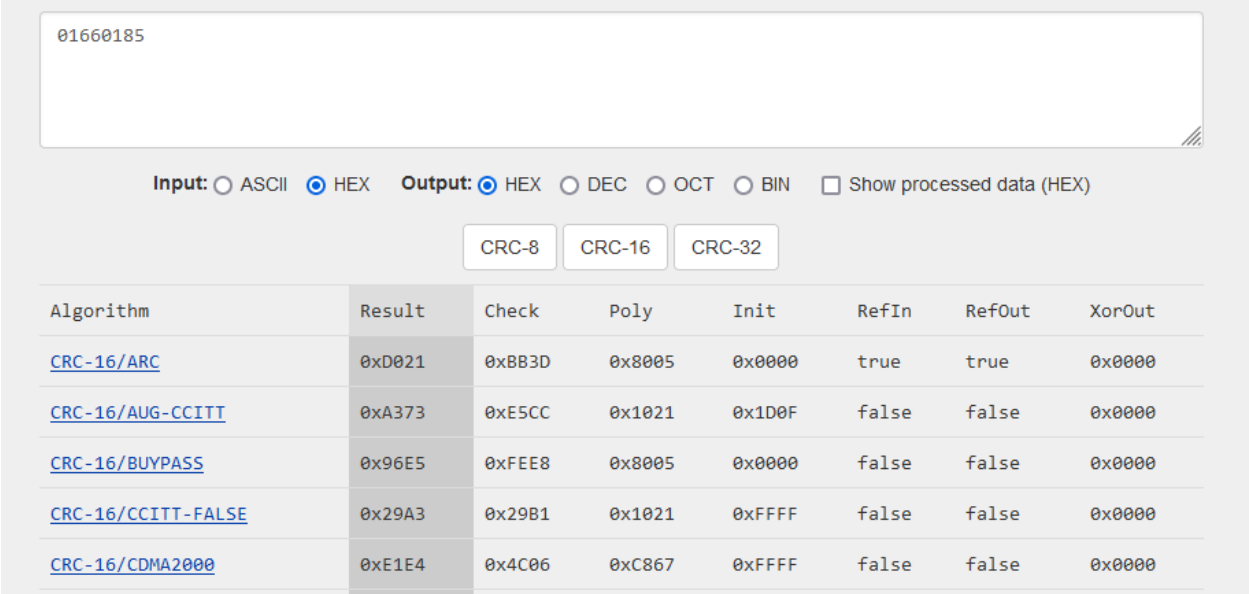
The uRAD payload transmits data over the serial port. A measurement of a radiation environment will consist of thousands of data points which uRAD sends in a continuous stream. A single data point consists of a pair of 12 bit numbers which are represented by 3 hexadecimal characters each. Each pair of numbers is followed by a 16 bit CRC. Thus, the total data size for any data point is 40 bits, or 10 hexadecimal characters. The use of the CRC makes the uRAD data set tolerant to corruption or interruption (a partial file can still be analyzed) at the cost of the fact that 40% of data volume is taken up by CRC data. With a firmware tweak, this can be reduced so the CRC is only sent for every 10th data point or so if desired.



A selection of data is shown in the window above from the uRAD serial interface application. The data shown does not represent the beginning of the file and thus nothing is known a-priori about where a particular data point ends and another starts. The CRC data can be used to separate out the different parts of a single datapoint. The same data from above is shown below.

EB1185166629A327B186D2101D417FBE852371144D11D416CA4B6267178F9FE1BA19F93DC
24E1EFB11C2F220D82C62BF1701A2A

Starting from the beginning of the selection of data, an algorithm moves forward (either hexadecimal character by hexadecimal character or bit by bit, if needed), calculates the CRC value for a given datapoint and compares it to the data in the position of the CRC value in the datapoint. The example below shows the first 12 bit value (the total integral value) highlighted in yellow and the second 12 bit value (for the “head” of the integral) highlighted in red. The CRC value in the dataset transmitted by uRAD is highlighted in green. CRC-16 algorithms expect a 16 bit input, so when the CRC was calculated, the 12 bit number was padded out to 16 bits by the addition of 4 leading zero bits. Thus, the first number shown below is “0185” and the second is “0166” for the purposes of the CRC calculation. When these numbers are put together (note the order in the CRC calculation is reversed from the order they appear in the transmitted data), the CRC calculation website shown below (https://crccalc.com/) produces the same CRC value as seen in the uRAD data (“29A3”), when using the CRC-16/CCITT-FALSE algorithm used by uRAD. A second example is also shown. As a result, it can be determined that the data highlighted in yellow and red represent an actual datapoint. If the same process is repeated with the positions shifted, the correct CRC value will not be the result.



Algorithm	Result	Check	Poly	Init	RefIn	RefOut	XorOut
CRC-16/ARC	0xD021	0xBB3D	0x8005	0x0000	true	true	0x0000
CRC-16/AUG-CCITT	0xA373	0xE5CC	0x1021	0x1D0F	false	false	0x0000
CRC-16/BUYPASS	0x96E5	0xFEE8	0x8005	0x0000	false	false	0x0000
CRC-16/CCITT-FALSE	0x29A3	0x29B1	0x1021	0xFFFF	false	false	0x0000
CRC-16/CDMA2000	0xE1E4	0x4C06	0xC867	0xFFFF	false	false	0x0000

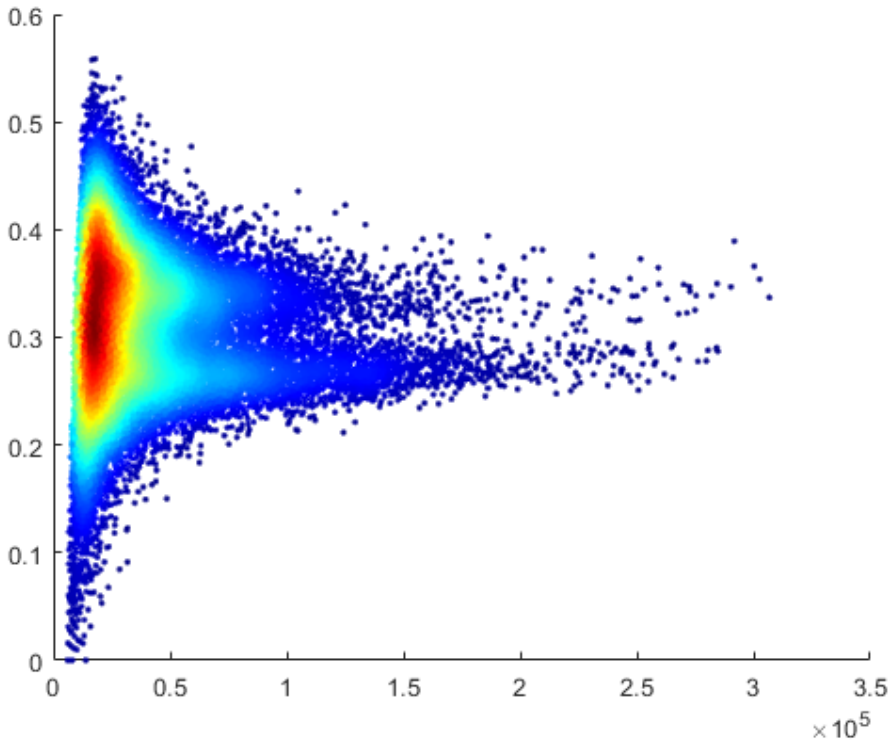
0186027B

Input: ○ ASCII ◉ HEX **Output:** ◉ HEX ○ DEC ○ OCT ○ BIN ☐ Show processed data (HEX)

CRC-8 | CRC-16 | CRC-32

Algorithm	Result	Check	Poly	Init	RefIn	RefOut	XorOut
CRC-16/ARC	0x56A1	0xBB3D	0x8005	0x0000	true	true	0x0000
CRC-16/AUG-CCITT	0x58C0	0xE5CC	0x1021	0x1D0F	false	false	0x0000
CRC-16/BUYPASS	0x9361	0xFEE8	0x8005	0x0000	false	false	0x0000
CRC-16/CCITT-FALSE	0xD210	0x29B1	0x1021	0xFFFF	false	false	0x0000
CRC-16/CDMA2000	0x1E90	0x4C06	0xC867	0xFFFF	false	false	0x0000

Once all of the CRC values have been checked and removed from the data set, it can be stored as a CSV file analysis. An example CSV file is included titled “uRAD_PSD.csv” as well as a MATLAB program that can produce a PSD diagram such as the one shown below. The text of the MATLAB program is also at the end of this document in case the file gets removed by JPL’s firewall.



```
file = 'uRAD_PSD.csv'
data = readmatrix(file);

%%

total = data(:,1);
head = data(:,2);

head=head*100+1;
total=total*100+1;

ratio = ((total-head)./total);
x = total;
y = ratio;


%%

colormap jet

%figure();
scatter_kde(x,y,".")% 'filled', 'MarkerSize', 100);

ylim([0 0.6]);

function h = scatter_kde(x, y, varargin)
% Scatter plot where each point is colored by the spatial density of nearby
% points. The function use the kernel smoothing function to compute the
% probability density estimate (PDE) for each point. It uses the PDE has
% color for each point.
%
% Input
%       x <Nx1 double> position of markers on X axis
%       y <Nx1 double> posiiton of markers on Y axis
%       varargin can be used to send a set of instructions to the scatter function
%       Supports the MarkerSize parameter
%       Does not support the MarkerColor parameter
%
% Output:
%       h returns handles to the scatter objects created
%
% Example
%       % Generate data
```

```
%       x = normrnd(10,1,1000,1);
%       y = x*3 + normrnd(10,1,1000,1);
%       % Plot data using probability density estimate as function
%       figure(1);
%       scatter_kde(x, y, 'filled', 'MarkerSize', 100);
%       % Add Color bar
%       cb = colorbar();
%       cb.Label.String = 'Probability density estimate';
%
% author: Nils Haentjens
% created: Jan 15, 2018
% Use Kernel smoothing function to get the probability density estimate (c)
c = ksdensity([x,y], [x,y]);
colormap jet
if nargin > 2
  % Set Marker Size
  i = find(strcmp(varargin, 'MarkerSize'),1);
  if ~isempty(i); MarkerSize = varargin{i+1}; varargin(i:i+1) = [];
  else MarkerSize = []; end
  % Plot scatter plot
  h = scatter(x, y, MarkerSize, c, varargin{:});
else
  h = scatter(x, y, [], c);
end
end
```