Developing a MOOS Driver for Marine Broadband Radar

Michael Sacarny, MIT Sea Grant College
Paul Robinette, MIT Sea Grant College

Marine Radar: "Ideal" requirements

- Range: A few meters to > 10 km
- Resolution: Can resolve objects (eg - pilings) near vessel
- Power requirements: The lower the better, active and standby
- Environmental: Fully marinized to rain, salt spray, snow, heat, cold
- Cost: < $2k
- Scan update rate: ~5 Hz or better
- Protocol: Public
- Software/Driver: Open source, Linux/Mac

Navico 4G Broadband radar

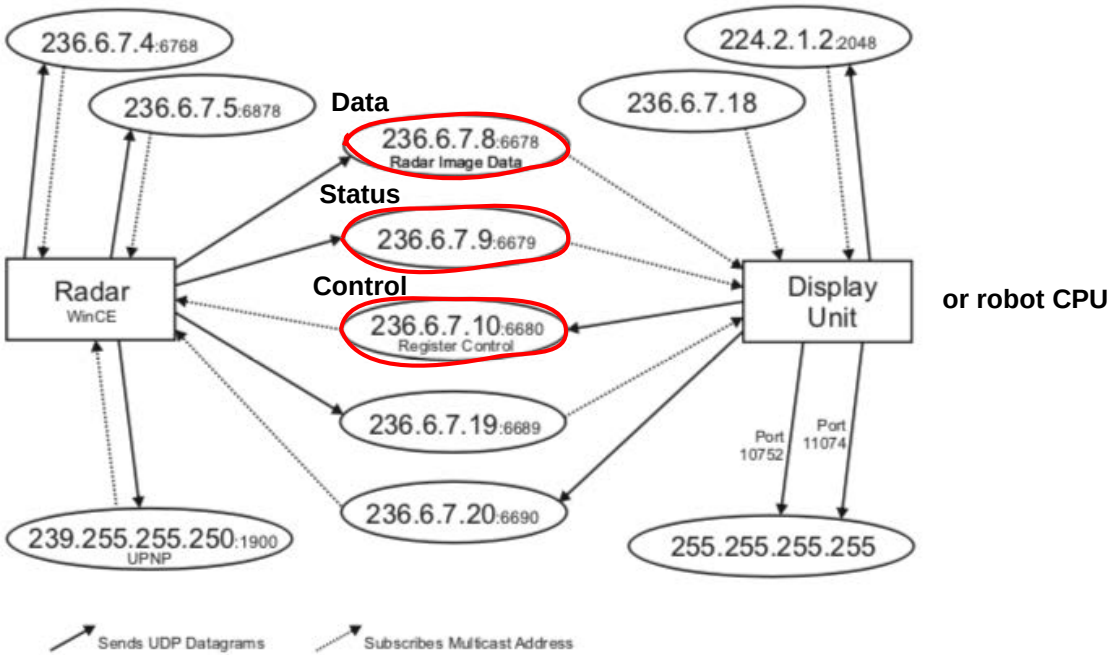
- ✓ Range: 50 m to 48 km
- ✓ Resolution: Can (sometimes) resolve pilings near vessel
- ✓ Power: 18 W active, 2 W standby (intensity ⅕ cell phone)
- ✓ Environmental: Fully marinized
- ✓ Cost: < $2k
- ~ Update rate: 48 rpm
- ✓ Bonus: Two channels!
- ✗ Protocol: Proprietary
- ✗ Software/Driver: Proprietary, data available on Navico Ethernet

Research: Past efforts

Dabrowski, Adrian, Sebastian Busch, and Roland Stelzer. "A digital interface for imagery and control of a Navico/Lowrance broadband radar." Robotic Sailing. Springer, Berlin, Heidelberg, 2011. 169-181.

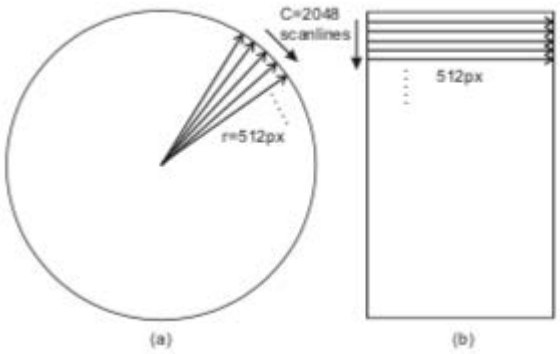
Test setup

Multicast data streams



From *Adrian Dabrowski, et. al.*

Radar data in packetized form



Radar imagery in polar form [2048 scanlines * 512 px] (a) and packetized form (b).

0	1	2	3	4	5	6	7	
01	00	00	00	00	20 ns	00 ss	02	Frame Header
18 l	02 st	73 rc	0B	00	44	0D	0E	Scanline Header
1D a	07	34	92	0C scale	00	00	01	
00	44	59 u1	13	00	40	00	CB u2	
512 polar pixels in 8 bit grayscale ⋮								Scanline

× 32

ns	Number of scan lines, always 32d (untested)
ss	Size of scan line, always 512d (untested)
l	length of scan line header, always 24d (untested)
st	unknown, probably status: 02 valid data; 18 spin up
rc	raw scan line counter 0..4095
a	absolute angle of scan line 0..4095 = 0..360°
scale	scan radius in meters = $scale * 10/\sqrt{2}$?
ux	variable, but unknown

From *Adrian Dabrowski, et. al.*

Protocol examination: Wireshark

Ethernet: en0

ip.dst == 236.6.7.8

No.	Time	Source	Destination	Protocol	Length	Info
1947	4.227038	192.168.1.2	236.6.7.8	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=14800, ID=7bda) [Reassembled in #1948]
1948	4.227042	192.168.1.2	236.6.7.8	UDP	922	39521 → 6678 Len=17160
1949	4.251543	192.168.1.2	236.6.7.8	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=0, ID=7bdb) [Reassembled in #1960]
1950	4.251558	192.168.1.2	236.6.7.8	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=1480, ID=7bdb) [Reassembled in #1960]
1951	4.251726	192.168.1.2	236.6.7.8	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=2960, ID=7bdb) [Reassembled in #1960]
1952	4.251867	192.168.1.2	236.6.7.8	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=4440, ID=7bdb) [Reassembled in #1960]
1953	4.252086	192.168.1.2	236.6.7.8	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=5920, ID=7bdb) [Reassembled in #1960]
1954	4.252197	192.168.1.2	236.6.7.8	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=7400, ID=7bdb) [Reassembled in #1960]
1955	4.252441	192.168.1.2	236.6.7.8	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=8880, ID=7bdb) [Reassembled in #1960]
1956	4.252509	192.168.1.2	236.6.7.8	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=10360, ID=7bdb) [Reassembled in #1960]
1957	4.252781	192.168.1.2	236.6.7.8	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=11840, ID=7bdb) [Reassembled in #1960]
1958	4.252987	192.168.1.2	236.6.7.8	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=13320, ID=7bdb) [Reassembled in #1960]
1959	4.253100	192.168.1.2	236.6.7.8	IPv4	1514	Fragmented IP protocol (proto=UDP 17, off=14800, ID=7bdb) [Reassembled in #1960]
1960	4.253207	192.168.1.2	236.6.7.8	UDP	922	39521 → 6678 Len=17160

```
▶ Frame 1: 1514 bytes on wire (12112 bits), 1514 bytes captured (12112 bits) on interface 0
▶ Ethernet II, Src: NavicoAu_0c:b2:a3 (00:0e:91:0c:b2:a3), Dst: IPv4mcast_06:07:08 (01:00:5e:06:07:08)
▶ Internet Protocol Version 4, Src: 192.168.1.2, Dst: 236.6.7.8
▼ Data (1480 bytes)
    Data: 9a611a164310edf001000000002000021802b60f00448000...
    [Length: 1480]
```

```
0020  07 08 9a 61 1a 16 43 10  ed f0 01 00 00 00 00 20   ··a··C· ·******
0030  00 02 18 02 b6 0f 00 44  80 00 72 07 00 80 a0 0c   *******D ··r·****
0040  b3 c0 00 00 00 80 00 00  00 a0 ff ff 00 00 00 00   ******** ********
0050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ******** ********
0060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ******** ********
0070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ******** ********
0080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ******** ********
0090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ******** ********
00a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ******** ********
00b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ******** ********
00c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ******** ********
00d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ******** ********
00e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ******** ********
00f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   ******** ********
```

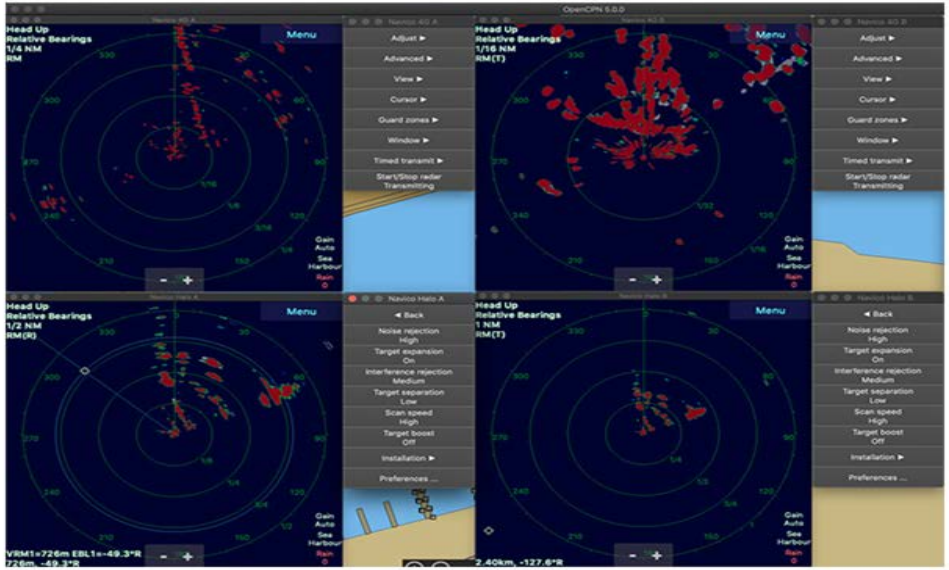
Data (data.data), 1480 bytes | Packets: 1960 · Displayed: 1956 (99.8%) | Profile: Default

Data Packets

Radar spoke data

Research: OpenCPN Plugin source code

Radar PI Plugin



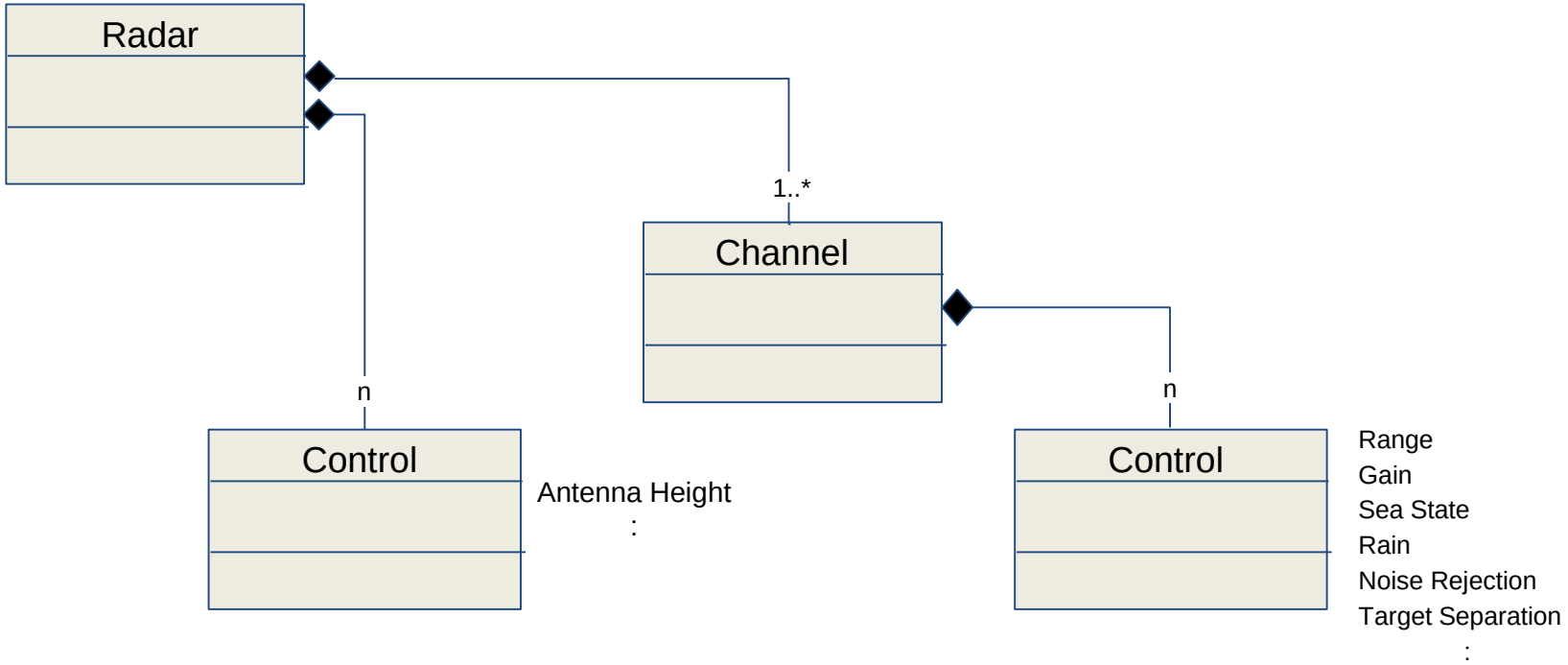
Challenges of reverse engineering

- You are at the mercy of the protocol
  - It could changed or be encrypted at any time
- Example: What determines the range of spoke data in the packet?
  - OpenCPN radar plugin discussion:

    "Why does the radar return such strange units for range in the packet header?"

    "Why?? Ask Navico. It is as it is. Here we just try to interpret."
  - Determined empirically, implemented as look-up table

Class Diagram



Implementation: MOOS class

Standard AppCastingMOOSApp class

```
class Radar : public AppCastingMOOSApp
{
 friend class Channel; // allow access to protected functions
 public:
   Radar();
   ~Radar();

 protected: // Standard MOOSApp functions to overload
   bool OnNewMail(MOOSMSG_LIST &NewMail);
   bool Iterate();
   bool OnConnectToServer();
   bool OnStartUp();

 protected: // Standard AppCastingMOOSApp function to overload
   bool buildReport();
   string buildChannelReport(int channel);
   void registerVariables();
   void notifySettings();
```

Implementation: MOOS configuration file

- Full details in Radar_Info.cpp

```
ProcessConfig = iRadar2
{
   AppTick          = 4
   CommsTick        = 4
   AntennaHeight    = 2.0
   0.run            = 1
   0.range          = 100
   0.gain           = Auto
   0.noiserej       = 3
   1.range          = 250
   1.gain           = 50

}
```

Implementation: Sockets and radar status

- I/O based on (cryptic) BOOST asio::ip::udp::socket classes
- Data passed between background and main thread by mutex-protected data queues
- Small state machine to handle overall radar channel status

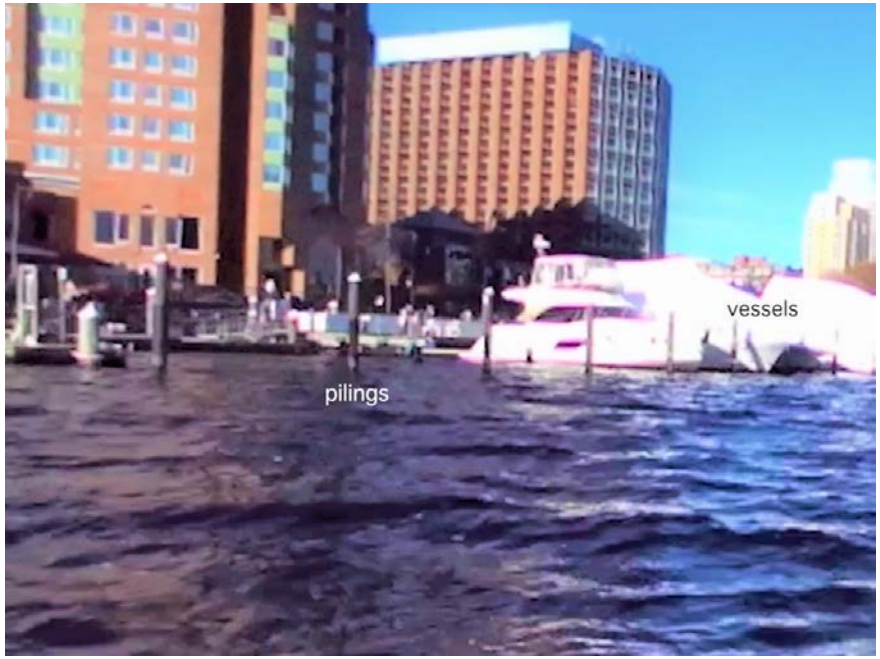
```
void Channel::changeStateChannel(int newState)
{
    stateChannelLast      = stateChannel;
    switch (newState) {
      case ST_CHANNEL_NO_COMMS:
      case ST_CHANNEL_NO_PULSE:
      case ST_CHANNEL_STANDBY:
      case ST_CHANNEL_SPINNING_UP:
      case ST_CHANNEL_TRANSMITTING:
      case ST_CHANNEL_WINDING_DOWN:
        stateChannel      = newState;
        break;
      default:
        parent->reportEvent(chDesc() + "In cha
        break;
    }
```

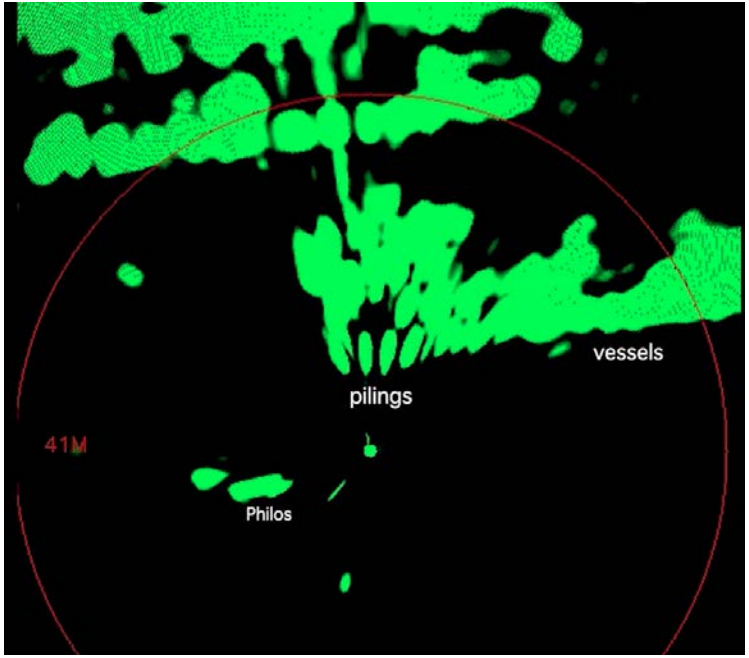
Implementation: MOOS DB variables

- Control changes through variable RADCMD
  - Notify (“RADCMD”, “0.gain = auto”)
- Status reports are available through variable RADRPT
  - RADRPT = 0.Range = 750
- Data packets available through variable RADRAW[0,1]
  - RADRAW0 = [spoke data]

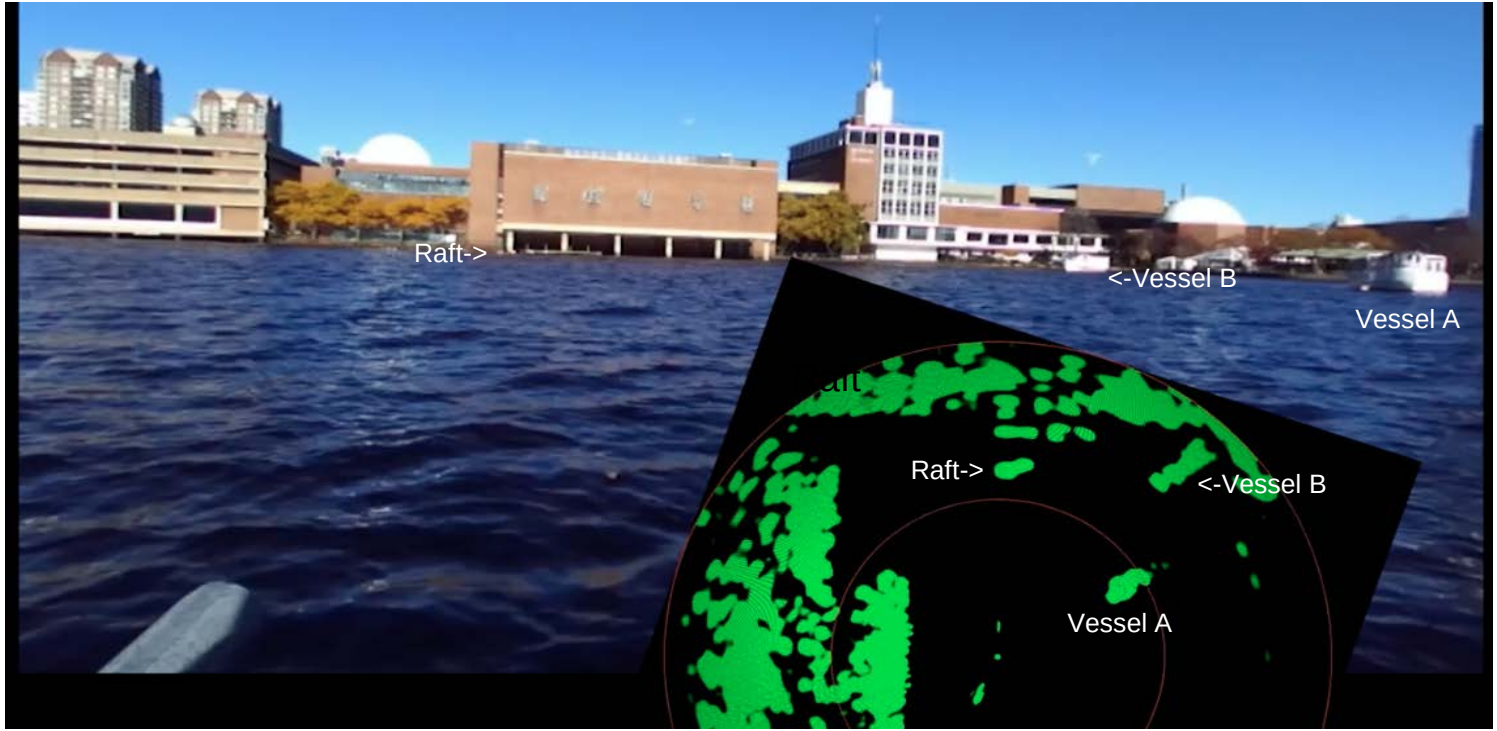
REx4 and Philos installations

Example: Yacht Club





Example: Museum of Science



Future work

- Better post-processing of data to reduce “blobbiness” and identify targets
- Algorithmic selection of optimal gain, range, and other control settings
- Exploit multiple radar channels to provide better overall coverage
- Fusion with other data sources
- Public release!

Acknowledgements

Many thanks to our corporate sponsor Brunswick Corporation and Mercury Marine!

BRUNSWICK

MERCURY