A MODULAR NETWORKED SYSTEM FOR COMBINED FLUIDIC, ELECTRONIC, AND THERMAL CONTROL OF A MULTIFUNCTIONAL RECONFIGURABLE ANTENNA ARRAY

A Thesis

by

NICHOLAS WILLIAM BRENNAN

Submitted to the Office of Graduate and Professional Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Chair of Committee,	Gregory H. Huff
Co-Chair of Committee,	Jean-Francois Chamberland-Tremblay
Committee Members,	Steven M. Wright
	John Valasek
Head of Department,	Chanan Singh

August 2014

Major Subject: Electrical Engineering

ABSTRACT

Recent work in the field of reconfigurable antennas has presented a variety of novel approaches to functionalizing antenna structures. In particular, fluidic & microfluidic strategies show promise as next-generation reconfiguration mechanisms to build advanced, highly-reconfigurable antenna designs capable of integration into cognitive wireless systems. In this work, a networked control system is conceptualized and implemented in a modular fashion to provide centralized control of an antenna array composed of such reconfigurable elements. A fluidic-controlled tri-band polarization & frequency reconfigurable antenna (TBPFRA) design—utilizing multiple fluid reconfiguration systems—is explored as a target design for control. An electronically polarization-reconfigurable antenna (EPRA) design is implemented and multifunctionalized with a thermoregulation system. The array control system is implemented on a seven element testbed platform with the multifunctional EPRA design. The assembled testbed system is then used to demonstrate a variety of cognitive antenna techniques, including beam steering and direction-of-arrival estimation. Finally, a novel method of raster-based infrared signaling is explored, and a proof-of-concept is demonstrated with the multifunctional array testbed.

DEDICATION

This thesis is dedicated to my parents, Frank & Donna. Without their love, encouragement, & support I would never have succeeded; with it I have accomplished more than I could have ever hoped.

ACKNOWLEDGEMENTS

I would first like to thank my advisor, Dr. Huff, for presenting me with this wonderfully challenging, interesting research problem. His guidance over the course of this project proved invaluable, and I consider myself lucky to have had the opportunity to work with him. Thanks also to my co-chair Dr. Chamberland, whose intriguing questions throughout my graduate career never failed to keep me on my toes.

I'd also like to extend a special thanks to all of my colleagues in the Texas A&M Electromagnetics & Microwave Laboratory. In particular, a special thanks to Jeffrey Jensen for his collaboration in the design of his phase shifter control board. Thanks also to David Grayson, Lisa Smith, and Deanna Sessions for their assistance in postprocessing and generating plots of the copious amounts of radiation pattern data I pulled out of this contraption, and to Sam Carey for his help with DoA measurements. Without their help, this thesis might have been a reasonable length!

Finally, I'd like to thank all of my friends—especially the ones who met me back in 2007 in Clements Hall and somehow decided they still wanted to hang around me—for both keeping me sane and keeping me goofy through this whole thing.

If I've only learned one thing from this whole endeavor, it's this: for any problem it's possible to find a solution so thoroughly incorrect that it actually works perfectly.

NOMENCLATURE

ρ	Bulk resistivity ($\Omega \cdot$ cm)
$\tan\delta$	Dielectric loss tangent
ε_r	Relative dielectric permittivity
ADC	Analog-to-digital converter
AESA	Active electronically-scanned array
ASCII	American Standard Code for Information Interchange
AWG	American wire gauge
BSTO	Barium strontium titanate
COSMIX	Coaxial stub microfluidic impedance transformer
DAC	Digital-to-analog converter
DHCP	Dynamic Host Configuration Protocol
DoA	Direction of Arrival
EFCD	Electromagnetically functionalized colloidal dispersion
eGaIn	Eutectic Gallium-Indium alloy
EPRA	Electronically polarization-reconfigurable antenna
I/Q	In-phase & Quadrature
IP	Internet Protocol
IR	Infrared
ISM	Industrial, scientific, and medical
LSB	Least-significant bit
LUT	Lookup table
MAC	Media access control
MCB	Modular control board

MCU	Microcontroller
MEMS	Micro-electromechanical systems
MSB	Most-significant bit
MUSIC	Multiple Signal Classification
NTC	Negative temperature coefficient
op-amp	Operational amplifier
PBSN	Polarization & band-switching fluid network
PESA	Passive electronically-scanned array
PID	Proportional-Integral-Derivative
PIN	Positive-Intrinsic-Negative
PLL	Phase-locked loop
SoC	System-on-a-chip
SPI	Serial peripheral interface
SSH	Secure shell
TBPFRA	Tri-band polarization- & frequency-reconfigurable antenna
TCP	Transmission Control Protocol
TCP/IP	Transmission Control Protocol/Internet Protocol
TEC	Thermoelectric Cooler/Thermoelectric Cooling
Thermal-IR	Electromagnetic radiation with wavelength ranging from 8μm–14μm
UART	Universal asynchronous reciever/transmitter
UI	User interface
VNA	Vector network analyzer

TABLE OF CONTENTS

Page

ABSTRACT ii

DEDICATION iii

ACKNOWLEDGEMENTS iv

NOMENCLATURE v

TABLE OF CONTENTS vii

LIST OF FIGURES x

LIST OF TABLES xv

1. INTRODUCTION 1

2. BACKGROUND 3

2.1 Microstrip Patch Antennas 3
2.1.1 Rectangular Microstrip Patch Antennas 4
2.1.2 Circular Microstrip Patch Antennas 13
2.2 Reconfigurable Patch Antennas 15
2.2.1 Electronic Reconfiguration Mechanisms 16
2.2.2 Fluidic Reconfiguration Mechanisms 18
2.3 Phased Antenna Array Control 22
2.4 Direction of Arrival Estimation 27
2.4.1 MUSIC Algorithm 28
2.5 Thermoelectric Cooling 30
2.6 Proportional-Integral-Derivative Control 33
2.6.1 Integral Anti-Windup 36
2.6.2 PID Tuning 39
2.7 TCP/IP Suite & Client-Server Architecture 43
2.7.1 TCP/IP Model 43
2.7.2 Client-Server Model 44
2.7.3 Transmission Control Protocol & Internet Sockets 45

3. TRI-BAND POLARIZATION- & FREQUENCY-RECONFIGURABLE ANTENNA (TBPFRA) 49
3.1 Design Goals 49
3.2 Design 50
3.2.1 Concept 50
3.2.2 Reconfiguration Mechanisms 54
3.3 Simulation 59
3.4 Fabrication & Testing 64
3.4.1 Electromagnetic Tests 64
3.4.2 Thermal Tests 66
3.4.3 Liquid Metal Tests 69

4. ELECTRONICALLY POLARIZATION-RECONFIGURABLE ANTENNA (EPRA) 72
4.1 Design Goals 72
4.2 Design 72
4.2.1 Concept 74
4.2.2 Reconfiguration Mechanism 74
4.2.3 Modeling & Simulation 79
4.3 Fabrication & Testing 81
4.3.1 Electromagnetic Tests 81

5. MODULAR CONTROL BOARD & THERMOREGULATION SYSTEM . 86
5.1 Design Goals 86
5.2 Design 87
5.2.1 Sensor Inputs 87
5.2.2 Reconfiguration Control Mechanisms 95
5.2.3 Processing, Communication, & Firmware 102
5.3 Fabrication & PID Algorithm Tuning 105

6. ARRAY CONTROL NETWORK & SERVER 109
6.1 Design Goals 109
6.2 Array Network Design 109
6.3 Control Server Implementation 110
6.3.1 Application Structure & Data Flow 111
6.3.2 UI Client Server 113
6.3.3 Module Connection Announcement Server 114
6.3.4 Module Command Handler 115
6.3.5 Network State Management 115
6.4 Server & Network Hardware 117

7. FULL MULTIFUNCTIONAL ARRAY SYSTEM EXPERIMENTAL RESULTS . . . 119
7.1 Thermoregulation & Thermal-IR Signaling Experiment . . . 121
7.1.1 Thermoregulation . . . 122
7.1.2 Thermal-IR Signaling . . . 123
7.2 Phased Array Beamforming Experiment . . . 128
7.3 Multiple Emitter Direction-of-Arrival Estimation Experiment . . . 133
8. SUMMARY . . . 138
8.1 Conclusion . . . 138
8.2 Future Work . . . 139
REFERENCES . . . 140
APPENDIX A. MCB FIRMWARE SOURCE CODE . . . 149
A.1 Main Code . . . 150
A.1.1 Powerup Initialization, Closed-Loop Temperature Control, & Main Loop Functions . . . 150
A.1.2 Discrete PID Control Algorithm . . . 155
A.2 Command Processing . . . 158
A.2.1 Command Parsing Library . . . 158
A.2.2 Command Execution Handler Functions . . . 161
A.3 Peripheral Control Code . . . 175
A.3.1 Analog/Digital Conversion & Temperature Conversion . . . 175
A.3.2 H-Bridge & Servo Control PWM Driver Libraries . . . 178
A.3.3 Phase Shifter Control DAC Driver Library . . . 184
A.4 Communication Code . . . 185
A.4.1 XBee WiFi Radio Initialization Code . . . 185
A.4.2 Serial String Communication Library . . . 188
A.4.3 SPI Master Controller Driver Library . . . 190
A.4.4 UART & USB Serial Port Driver Libraries . . . 191
APPENDIX B. ARRAY CONTROL SERVER SOURCE CODE . . . 211
B.1 Main Code . . . 212
B.2 User Interface Server . . . 217
B.3 Antenna Module Servers . . . 219
APPENDIX C. MODULE COMMAND REFERENCE . . . 223
APPENDIX D. MODULAR CONTROL BOARD DESIGN & PCB LAYOUT 225

LIST OF FIGURES

FIGURE Page

2.1 Geometry & principal design variables of a rectangular microstrip patch antenna 4

2.2 Cross-section of rectangular microstrip patch antenna 5

2.3 Transmission line model of rectangular microstrip patch antenna . . . 6

2.4 Electric field distribution of a rectangular microstrip patch antenna . 10

2.5 Geometry & behavior of the COSMIX 23

2.6 Uniform linear phased antenna array with corporate feed network from common RF source 25

2.7 Uniform linear phased antenna array with multiple phase-locked transmitters 26

2.8 Direction of arrival estimation using multiple phase-locked receivers . 28

2.9 Construction of a Peltier thermoelectric cooling device 31

2.10 Theory of operation of a Peltier TEC device 33

2.11 Canonical interacting PID controller 34

2.12 Interacting PID controller with limited control output & conditional integration 38

2.13 Measured thermal control loop step response with process deadtime t_d annotated 42

2.14 Measured thermal control loop step response with time constant τ and gain $g_p = \Delta y / \Delta u$ annotated 42

2.15 TCP/IP layered data flow model 45

2.16 TCP connection establishment process (right) and connection teardown process (left) 48

3.1 Concentric circular patches analogous to TBPFRA 50
3.2 Switching bands by filling gaps 51
3.3 TBPFRA metallization top view 52
3.4 TBPFRA design parameters 53
3.5 TBPFRA polarization & band switching network fluid channel layout 55
3.6 PBSN control & sensing schematic (each fluid channel, 2 total) 56
3.7 TBPFRA operating modes 57
3.8 TBPFRA model front with COSMIX 58
3.9 TBPFRA model rear with COSMIX 58
3.10 COSMIX fluid control network for TBPFRA 59
3.11 TBPFRA high-band simulation: tuning ε_r in inner & outer x-arm COSMIX elements 60
3.12 TBPFRA mid-band simulation: tuning ε_r in inner x-arm COSMIX elements 62
3.13 TBPFRA low-band simulation: tuning ε_r in inner & outer x-arm COSMIX elements 63
3.14 Fabricated single-arm TBFPRA prototype 65
3.15 Analog phase shifters used to emulate COSMIX 66
3.16 TBPFRA prototype operation without & with phase shifters 67
3.17 Measured TBPFRA with phase shifters, tuning mid-band mode . . . 68
3.18 TBPFRA heat exchanger design 68
3.19 Thermal control system testing at 25°C ambient temperature 69
3.20 eGaIn plug in 1000 cst silicone oil showing debris sloughing 70
3.21 Optical microscopy of eGaIn debris in silicone oil 71
4.1 EPRA design concept 73
4.2 DC bias controls EPRA polarization state 75

4.3 EPRA DC biasing network 77

4.4 DC bias operation (red: positive, blue: negative) 78

4.5 EPRA element *HFSS* model 79

4.6 Simulated EPRA impedance behavior: x-polarized mode 80

4.7 Simulated EPRA radiation patterns: x-polarized mode 81

4.8 Fabricated EPRA element mounted in hexagonal array 82

4.9 Return loss of fabricated EPRA elements 83

4.10 Input impedance of fabricated EPRA elements (measured from 2GHz–3GHz, normalized to 50Ω) 84

4.11 Measured EPRA element radiation pattern: x-polarized mode 85

4.12 Measured EPRA element radiation pattern: y-polarized mode 85

5.1 Block diagram of modular control board 87

5.2 Lateral (left) and axial (right) cross-sections of a conductive fluid sensor in a fluid tube 88

5.3 Fluid sensor circuit with only dielectric fluid present 89

5.4 Fluid sensor circuit with passing liquid metal plug 90

5.5 Thermistor temperature sensor circuit diagram 91

5.6 Resistance vs. temperature for Vishay 01M1002KF NTC thermistor . 92

5.7 ADC word (hexadecimal) vs. temperature for implemented thermistor circuit 94

5.8 4x 3-way servo-actuated fluidic valve network 95

5.9 H-bridge circuit controlling current through a TEC 97

5.10 Hittite Microwave HMC928LP5E 2–4GHz 0–450° analog phase shifter 98

5.11 Block diagram of phase shifter control DAC card 99

5.12 Phase shifter control DAC card 99

5.13 Block diagram of antenna element thermoregulation system 100

5.14 Assembled Peltier TEC heat pump 101

5.15 MCU firmware block diagram 104

5.16 Fabricated modular controller board 106

5.17 Tuned PID temperature controller test results 108

6.1 Array control network structure 110

6.2 Block diagram of server application design 112

6.3 *Raspberry Pi* array control server (left) and *Carambola* WiFi router mounted on array testbed 117

7.1 Assembled multifunctional reconfigurable antenna array 120

7.2 Thermoregulation test experimental setup 121

7.3 Array-wide thermoregulation 122

7.4 Logical numbering of EPRA elements in the hexagonal array 124

7.5 Mapping ASCII-encoded bits to antenna elements 124

7.6 ASCII character signaling using element temperatures 127

7.7 Thermal signaling character timing 128

7.8 Amplitude & phase balance of phased array corporate feed network . 129

7.9 Measured normalized array pattern: x-polarized mode, in-plane steering 130

7.10 Measured normalized array pattern: y-polarized mode, in-plane steering 130

7.11 Measured normalized array pattern: x-polarized mode, out-of-plane steering 131

7.12 Measured normalized array pattern: y-polarized mode, out-of-plane steering 132

7.13 Conceptual overview of multiple emitter resolution using polarization reconfiguration 132

7.14 Polarization-reconfigurable direction-of-arrival estimation experimental setup 134

7.15 MUSIC pseudospectra: x-polarized array 135

7.16 MUSIC pseudospectra: y-polarized array 136
D.1 Modular control board: main schematic 226
D.2 Modular control board: H-bridge circuitry 227
D.3 Modular control board: servo valve controls 228
D.4 Modular control board: sensor inputs 229
D.5 Modular control board PCB: bottom copper 230
D.6 Modular control board PCB: top copper 231
D.7 Modular control board PCB: silkscreen 232
D.8 Modular control board PCB: soldermask 233

LIST OF TABLES

TABLE Page

2.1 Summary of liquid metal properties 19

2.2 Dielectric properties of selected dielectric fluids 20

3.1 IEEE standard letter designations for radar frequency bands 49

6.1 Command structure for UI client → control server communication . . 114

7.1 Temperature representation of bit values 125

7.2 Comparison of MUSIC-estimated and physically measured direction of arrival 137

C.1 Modular controller board command reference 224

1. INTRODUCTION

Wireless data transmission has become ubiquitous in modern electronic devices. So, too, has the number of wireless data transmission standards proliferated. Increasingly, mobile devices are expected to communicate using multiple wireless standards, and in multiple frequency bands. Furthermore, many commercial & military users require both land-mobile and satellite-mobile wireless communications for short-range and long-range communication, respectively. This multitude of requirements plays to the strengths of a variety of different canonical antenna designs, however achieving high gain, wide operating bandwidth, and polarization diversity in a single passive antenna design is quite difficult.

Reconfigurable antennas show promise to provide communications system designers with the ability to implement truly multifunctional communications systems. Frequency reconfigurability can allow a single antenna system to work in multiple different frequency bands, and polarization reconfigurability allows an antenna system to leverage polarization diversity to combat signal fading and multipath effects. A variety of different techniques are available to reconfigure the operating behavior of an antenna element. In particular, a variety of solid-state (and near-solid-state) electronic mechanisms can be used to reconfigure the radiation pattern, operating frequency, and polarization of an antenna structure. Recently, microfluidic systems have shown promise as an alternative to electronic reconfiguration mechanisms for antenna applications. Fluidic mechanisms have the potential to enable higher RF power operation, with lower loss than comparable electronic mechanisms, and without requiring conductive control wiring on the antenna structure—obviating the potential such control structures have to perturb the operation of the antenna.

In stationary and mobile applications, antenna arrays can be used to provide higher gain and a more directional radiation pattern through beamforming, allowing further improvement of the signal-to-noise ratio and link budget of a wireless link. Furthermore, when equipped with controllable phase shifting elements and phase-sensitive receivers, such an antenna array can be used to implement electronically-steerable beamforming and direction-of-arrival estimation capabilities. Such a mobile or stationary array could then be used to locate and track a remote transceiver. An antenna & transceiver array system like this has many applications, ranging from multiuser wireless communications systems to radar to electronic warfare.

This thesis will explore the preliminary development of a multi-band, frequency- and polarization-reconfigurable planar antenna, using novel fluidic reconfiguation techniques, which can be tiled in a hexagonal array for distributed beamforming and direction-of-arrival estimation applications. In particular, the control system to manipulate the reconfiguration mechanisms on an array of such antenna elements is developed and tested. The control system utilizes wired & wireless TCP/IP networking to implement a dynamically reconfigurable array control system, allowing individual antenna elements to be added and removed from the array on the fly.

To achieve high gain with an antenna array, many elements are required. For planar antenna elements in a planar array, a high gain equates to a large planar surface, which typically must be unimpeded by external structures to avoid compromising the RF performance of the array. This thesis will further explore the multifunctionalization of such a planar antenna array through the implementation of a thermoregulation system. By achieving individual control of each antenna element's temperature, a multifunctional array system capable of displaying a long-wave infrared image is achieved. A particular envisioned application is explored: the use of a such a multifunctional array to transmit data via long-wave infrared energy.

2. BACKGROUND

2.1 Microstrip Patch Antennas

Microstrip patch antennas are a common modern antenna structure, useful in a variety of applications where size, weight, and cost are key design constraints [1]. Microstrip antennas are easily conformable to the surface of a wide variety of structures, are readily and easily manufactured using the same techniques used for planar circuit fabrication, and are amenable to the addition of a variety of reconfiguration mechanisms to enable manipulation of their operating frequency, impedance, polarization, and/or radiation pattern. [2, 3] Microstrip patch antennas can also be constructed in a wide variety of geometries. Common to all microstrip antennas are four key elements [4]:

- a thin conductive sheet—usually metallic—called the patch
- a (typically larger) conductive sheet known as the ground plane
- a dielectric substrate separating the two conductive sheets, and
- a feed structure, which couples electromagnetic energy into the antenna

Despite their numerous advantages, microstrip patch antennas have several key disadvantages. The principal disadvantage of a patch antenna is its high quality factor, Q. The Q of a structure or circuit can be expressed as

$$Q = 2\pi \frac{\text{Energy stored}}{\text{Energy lost per cycle}} \tag{2.1}$$

As will be examined in depth, patch antennas are roughly resonant structures, so the *Energy stored* term in (2.1) is typically large relative to the *Energy lost per cycle*

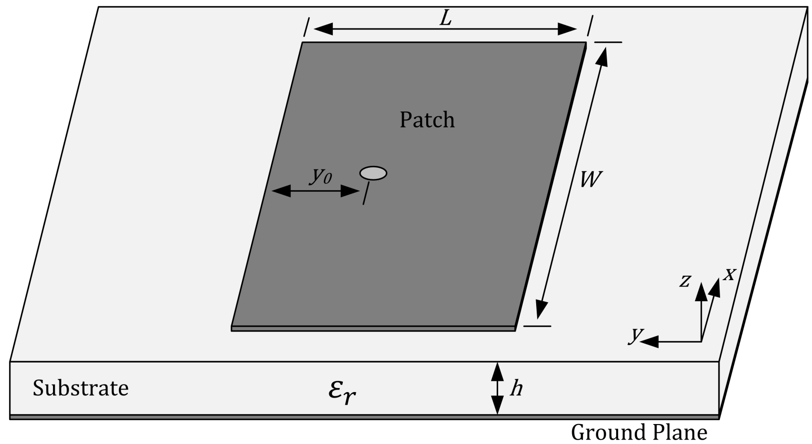


Fig. 2.1: Geometry & principal design variables of a rectangular microstrip patch antenna

term. The high Q of most patch antennas results in several disadvantageous effects. First, patch antennas are typically narrowband structures, with small impedance bandwidths of at most a few percent. Microstrip antennas also typically exhibit low radiation efficiencies as only a fraction of the energy supplied into them is lost through the radiating mechanism. Although approaches exist to reduce the Q of a microstrip antenna, most result in a degradation of the radiation pattern and/or polarization of the antenna. Despite these disadvantages, microstrip antennas have seen significant use in applications where weight, profile, and cost constraints are tight. Below, two of the most common patch geometries—from which the antenna geometries explored in this work are derived—are examined.

2.1.1 Rectangular Microstrip Patch Antennas

2.1.1.1 Overview

The rectangular microstrip patch antenna is the canonical form of the microstrip patch, and the first geometry explored in literature [5]. Fig. 2.1 shows an overview of the geometry of the rectangular patch and its key design variables. The antenna takes

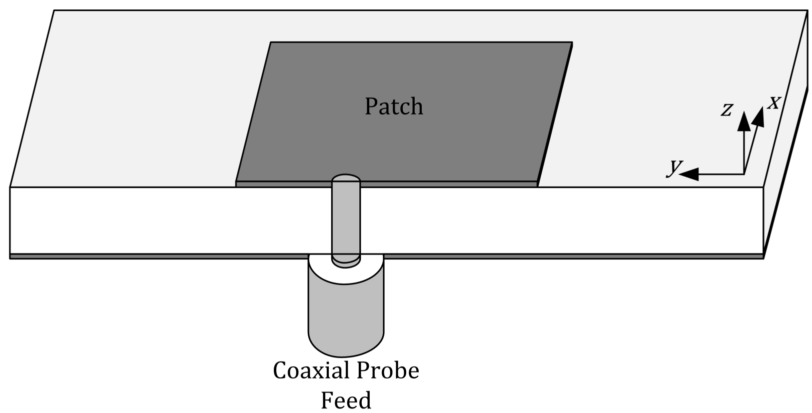


Fig. 2.2: Cross-section of rectangular microstrip patch antenna

the form of a thin rectangular patch, laid atop a substrate material of thickness h with some relative permittivity ε_r. This substrate material is underlain by a conducting metal ground plane. The rectangle of the patch is defined by a length L, and a width W. These three variables–L, W, and h–determine the operating frequency, impedance & operating bandwidths, and radiation pattern of the patch antenna.

A rectangular patch antenna is typically designed such that the peak of its radiation pattern is normal to the plane of the patch itself (the $+z$ direction in Fig. 2.1). This is accomplished with a geometry in which the substrate thickness is small relative to the operating wavelength ($h << \lambda_0$), and the resonant length is chosen such that $\lambda_0/3 < L < \lambda_0/2$ [1].

2.1.1.2 Feed Mechanisms

A common method of feeding electromagnetic energy into the patch is the coaxial probe feed, a cross-section of which is shown in Fig. 2.2. Note how the center conductor of the coaxial probe is connected to the patch, and the outer conductor is coupled to the ground plane. A hole of radius equal to that of the dielectric space in the coaxial cable is cut into the ground plane to facilitate the passage of

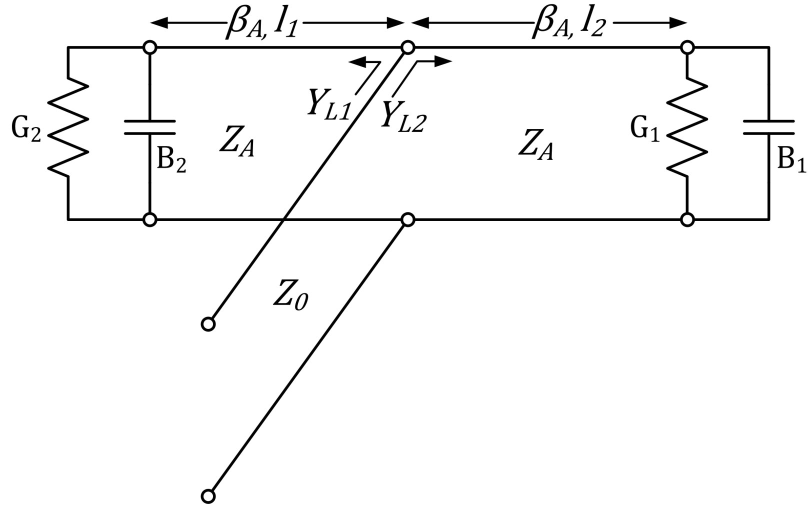


Fig. 2.3: Transmission line model of rectangular microstrip patch antenna

the electromagnetic fields into the dielectric beneath the patch. The distance y_0 at which the probe is inset into the patch from the radiating edge on the $+y$ side of the patch controls the input impedance of the patch seen at the probe feed, allowing the feed to be impedance matched to the antenna. Other common feed geometries include inset microstrip feed lines, aperture coupled microstrip feeds, and proximity coupled microstrip feeds. Only the coaxial probe feed geometry was explored in this work.

2.1.1.3 Analysis & Design

Two analytical models are commonly used to gain insight into the operation and design of the rectangular patch: the *transmission-line model* and the *cavity model.* The transmission-line model was one of the first analytical models developed for the patch antenna [5], and while it does not yield the most accurate results for operating parameters such as frequency and input impedance, it does provide some insight into the operating behavior of the patch antenna. The cavity model of the patch antenna is slightly more complex, but provides more accurate predictions of the operating frequency and input impedance. Further, the cavity model provides more physical insight into the radiation mechanism of the patch, and a reasonably accurate approximation of its radiation pattern.

Fig. 2.3 shows a schematic representation of the transmission line patch model. In this model, the patch antenna is modeled as a section of wide, low-impedance microstrip transmission line of characteristic impedance Z_A, propagation constant β_A, and length $l_1 + l_2$. The characteristic impedance of a microstrip transmission line is determined primarily by its width w and the thickness of the dielectric substrate h on which it rests, along with the relative permittivity ε_r of the substrate. For microstrip line, this characteristic impedance is given by: [6]

$$Z_0 = \begin{cases} \dfrac{60}{\sqrt{\varepsilon_{\text{eff}}}} \ln\left(\dfrac{8h}{w} + \dfrac{w}{4h}\right) & \text{for } \dfrac{w}{h} \leq 1, \\ \dfrac{120\pi}{\sqrt{\varepsilon_{\text{eff}}}} \left[\dfrac{w}{h} + 1.393 + 0.667 \ln\left(\dfrac{w}{h} + 1.444\right)\right]^{-1} & \text{for } \dfrac{w}{h} > 1 \end{cases} \tag{2.2}$$

The term ε_{eff} in (2.2) represents an effective relative permittivity which, if it replaced the dielectric of the microstrip line *and the air above it* such that the microstrip were embedded in a uniform dielectric, would result in a transmission line with electrical properties identical to the actual geometry. This term is given by [6]

$$\varepsilon_{\text{eff}} = \frac{\varepsilon_r + 1}{2} + \frac{\varepsilon_r - 1}{2}\left[1 + 12\frac{h}{w}\right]^{-1/2} \quad \text{for } \frac{w}{h} > 1 \tag{2.3}$$

In the transmission line model of the microstrip patch, the impedance given by (2.2) is substituted for Z_A. β_A can be found from

$$\beta_A = \frac{2\pi}{\lambda_0}\sqrt{\varepsilon_{\text{eff}}} \tag{2.4}$$

where λ_0 is the free-space wavelength and ε_{eff} is given by (2.3). The electric fields between the patch and ground plane take the form shown in Fig. 2.4, derived from the cavity model of the patch. Because the patch geometry is finite, the electric field between the patch and ground plane fringe outward at the edges. This fringing also occurs at the boundaries of the patch in the orthogonal cut-plane (x-z plane) as well. These fringing fields result in an effective electrical length & width extension of the patch. The effective width can be approximated as [7]

$$w_{\text{eff}} = \frac{120\pi h}{Z_m\sqrt{\varepsilon_{\text{eff}}}} \tag{2.5}$$

$$Z_m = \frac{60\pi}{\sqrt{\varepsilon_{\text{eff}}}}\left[\frac{w}{2h} + 0.441 + \frac{1.451}{\pi} + \ln\left(\frac{w}{2h}\right) + 0.94\right]^{-1} \tag{2.6}$$

Similarly, the effective length extension ΔL, such that the effective length $L_{\text{eff}} = L + 2\Delta L$ can be found from [8]

$$\Delta L = 0.412\frac{(\varepsilon_{\text{eff}} + 0.3)\left(\frac{w}{h} + 0.264\right)}{(\varepsilon_{\text{eff}} - 0.258)\left(\frac{w}{h} + 0.8\right)}h \tag{2.7}$$

The operating frequency of the patch can be then be estimated according to [1]

$$f_c = \frac{1}{2L_{\text{eff}}\sqrt{\varepsilon_{\text{eff}}}\sqrt{\mu_0\varepsilon_0}} = \frac{1}{2\left(L + 2\Delta L\right)\sqrt{\varepsilon_{\text{eff}}}\sqrt{\mu_0\varepsilon_0}} \tag{2.8}$$

The radiating slots at the edges of the patch—denoted as the *fringing fields* in Fig. 2.4—can be represented in the transmission-line model as a complex admitttance $Y_{1,2} = G_{1,2} + jB_{1,2}$, where $G_{1,2}$ represents the conductance of slots 1 & 2, respectively, due to radiation loss. $B_{1,2}$ represents the capacitance of the fields in the slot. Several approximations of varying accuracy exist to evaluate these slot admittances. The simplest—though not necessarily most accurate—is that based on a slot of infinite width, where [1]

$$G_{1,2} = \frac{W}{120\lambda_0}\left[1 - \frac{1}{24}(k_0 h)^2\right] \qquad \frac{h}{\lambda_0} < \frac{1}{10} \tag{2.9}$$

$$B_{1,2} = \frac{W}{120\lambda_0}\left[1 - 0.636\ln(k_0 h)\right] \qquad \frac{h}{\lambda_0} < \frac{1}{10} \tag{2.10}$$

λ_0 is the free space wavelength at the operating frequency and $k_0 = \frac{2\pi}{\lambda_0}$. The input admittances Y_{L1} and Y_{L2} in Fig. 2.3 can be found from

$$Y_{\text{L1,L2}} = Y_A \frac{Y_{1,2} + jY_A \tan(\beta_A l_{1,2})}{Y_A + jY_{1,2}\tan(\beta_A l_{1,2})} \quad \text{where } Y_A = \frac{1}{Z_A} \tag{2.11}$$

The input impedance of the patch antenna at its operating frequency can thus be calculated according to the following process

1. Use the effective width from (2.5) in (2.2) to calculate Z_A in Fig. 2.3
2. Calculate the propagation constant β_A from (2.4)
3. Use (2.10) to calculate $Y_{1,2} = G_{1,2} + jB_{1,2}$ in Fig. 2.3
4. Calculate effective line lengths $l_{1,\text{eff}} = l_1 + \Delta L$, $l_{2,\text{eff}} = l_2 + \Delta L$ using (2.7)
5. Calculate the input admittances Y_{L1} and Y_{L2} of the two halves of the patch according to (2.11)
6. Calculate the input impedance of the patch antenna as $Z_{\text{in}} = 1/\left(Y_{L1} + Y_{L2}\right)$

The cavity model of the microstrip patch antenna is slightly more complex than the transmission-line model, but it tends to give a more accurate estimate of the operating frequency. Further, the cavity model directly incorporates insight into the radiating mechanism of the patch, and thus provides a fairly accurate estimate of its radiation pattern.

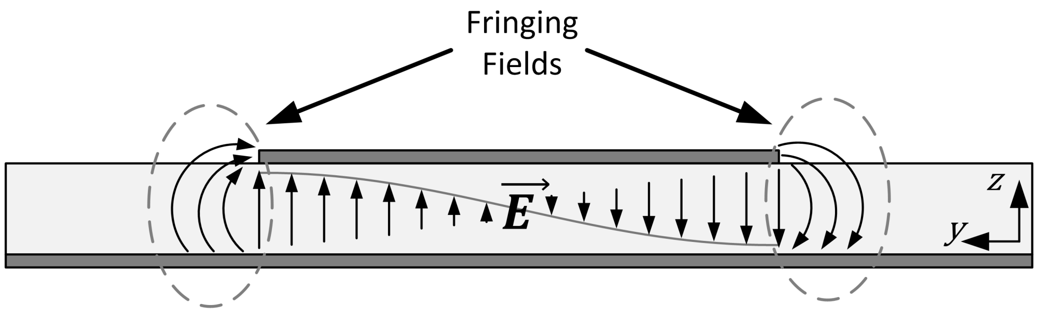


Fig. 2.4: Electric field distribution of a rectangular microstrip patch antenna

The cavity model is based on the assumption that the region of the dielectric substrate between the patch and ground plane can be treated as a resonant cavity. To simplify the analysis, the cavity surfaces bounded by the metal patch and ground plane are treated as perfectly electric conducting boundaries (with zero tangential electric fields). Likewise, by considering the current flow around the edges of the patch from the bottom surface to the top surface, an approximation can be made to treat the vertical cavity boundaries around the perimeter of the patch as perfect magnetic conductors (with zero tangential magnetic fields). [1]

Because the height h of the cavity is typically very small with respect to the operating wavelength ($h << \lambda$), a reasonable approximation is to treat the electric field below the patch as perfectly normal to the conductor surfaces. With this assumption, only modes with magnetic fields transverse to z are considered. The fields are found by solving the homogeneous wave equation for the magnetic vector potential

$$\nabla^2 A_z + k^2 A_z = 0 \tag{2.12}$$

whose general solution is [6]

$$
\begin{aligned}
A_z =& [A_1 \cos(k_x x) + B_1 \sin(k_x x)] \cdot \\
& [A_2 \cos(k_y y) + B_2 \sin(k_y y)] \cdot \\
& [A_3 \cos(k_z z) + B_3 \sin(k_z z)]
\end{aligned}
\tag{2.13}
$$

By relating A_z to the electric fields in the cavity and subjecting those fields to the boundary conditions of the cavity model (zero tangential electric fields at the top and bottom boundaries, zero tangential magnetic fields on the vertical perimeter boundaries) this solution becomes [6]

$$A_z = A_{mnp} \cos(k_x x') \cos(k_y y') \cos(k_z z') \tag{2.14}$$

at a point (x', y', z') inside the cavity, where A_{mnp} is the amplitude coefficient of the *mnp* mode and

$$k_x = \frac{m\pi}{W}, \qquad m = 0, 1, 2, \dots \tag{2.15a}$$

$$k_y = \frac{n\pi}{L}, \qquad n = 0, 1, 2, \dots \tag{2.15b}$$

$$k_z = \frac{p\pi}{h}, \qquad p = 0, 1, 2, \dots \tag{2.15c}$$

are the wavenumbers in the x, y, and z directions respectively. m, n, and p are the mode numbers for the respective axes, and can take any set of integer values *except* $m = n = p = 0$.

The resonant frequency of the cavity modes can be found by substituting (2.15) into the constraint

$$k_x^2 + k_y^2 + k_z^2 = k_r^2 = \omega_r^2 \mu \varepsilon \tag{2.16}$$

which gives the resonant frequency for the *mnp* mode as

$$f_{r,mnp} = \frac{1}{2\pi\sqrt{\mu\varepsilon}}\sqrt{\left(\frac{m\pi}{W}\right)^2 + \left(\frac{n\pi}{L}\right)^2 + \left(\frac{p\pi}{h}\right)^2} \tag{2.17}$$

Typically, the dimensions L and W are chosen such that the lowest order mode is the TM^z_{010} mode, such that the resonant direction of the patch in Fig. 2.1 is along the y-axis. When calculating the resonant frequency of the patch cavity using (2.17), it is best to reincorporate the effects of the fringing fields around the periphery of the patch that were approximated out during the derivation of the cavity model. This can be accomplished by substituting an effective length L_{eff} and width W_{eff} into (2.17) using the microstrip length & width extension formulas in (2.7) & (2.5). This will give an improved estimation of the operating frequency of the patch.

The cavity model also gives us mathematical tools to predict the radiation pattern of the rectangular patch with reasonable accuracy. By applying Huygens' equivalence principle to the fields of the cavity model, the fields of the cavity and the physical structure of the patch can be replaced by an equivalent magnetic surface current $\mathbf{M}_s$ where

$$\mathbf{M}_s = -2\hat{\mathbf{n}} \times \mathbf{E}_a \tag{2.18}$$

In this equivalence expression, $\mathbf{E}_a$ is the electric field vector at the vertical (PMC) boundary surrounding the patch, and is multiplied by 2 to account for the equivalent current image in the ground plane below the patch. Along the resonant length L of the patch, the electric field is equal amplitude and opposite on the $+y$ and $-y$ sides as shown in Fig. 2.4, so the equivalent magnetic currents along these edges are zero and these slots are considered *nonradiating*. Along the nonresonant width W of the patch, the electric field is uniform and nonzero, and by applying (2.18) to the two walls with opposite normal vectors $\hat{\mathbf{n}}$ one finds that the equivalent magnetic currents $\mathbf{M}_{s,1}$ and $\mathbf{M}_{s,2}$ are equal and in phase. Thus, the patch can be simplified to a two-element array of magnetic currents which radiate with a maximum at broadside ($+z$). The far-field electric field radiated by each current can be expressed as [1]

$$E_r \approx E_\phi \approx 0 \tag{2.19a}$$

$$E_\theta \approx j\frac{k_0 h W E_0 e^{-jk_0 r}}{2\pi r}\left[\cos\theta \frac{\sin Z}{Z}\right] \tag{2.19b}$$

where

$$Z = \frac{k_0 W}{2}\sin\theta\cos\phi \tag{2.19c}$$

for $k_0 h << 1$ By applying array theory, an array factor AF can be calculated for the two slots separated by a distance L_{eff} in the y direction as

$$AF_y = 2\cos\left(\frac{k_0 L_{\text{eff}}}{2}\sin\theta\sin\phi\right) \tag{2.20}$$

which can be applied to (2.19b) to find the total radiated electric field: [1]

$$E_\theta \approx j\frac{k_0 h W E_0 e^{-jk_0 r}}{\pi r}\left[\cos\theta \frac{\sin\left(\frac{k_0 W}{2}\sin\theta\cos\phi\right)}{\frac{k_0 W}{2}\sin\theta\cos\phi}\right] \times \cos\left(\frac{k_0 L_{\text{eff}}}{2}\sin\theta\sin\phi\right) \tag{2.21}$$

2.1.2 Circular Microstrip Patch Antennas

A circular microstrip patch antenna has a fundamentally similar geometry to that shown in Fig. 2.1, with the exception that the boundary of the patch on the top surface of the dielectric substrate is defined—instead of by a rectangle of dimensions L and W—by a circle of radius r centered about the z-axis (i.e. a metal circle in the $x - y$ plane centered at $(x, y, z) = (0, 0, h)$). The geometry of the circular patch is only readily amenable to the application of the cavity model described above, as the transmission-line model does not adapt well to this geometry.

As with the rectangular patch, the dielectric between the patch and ground plane is treated as a cavity—a cylindrical cavity—with circular PEC top and bottom surfaces and a PMC wall bounding the cylindrical wall. The analysis proceeds as in section 2.1.1.3. The fields are assumed to be TM_z, such that the electric field in the cavity is normal to the top and bottom PEC walls. (2.12) is solved for the magnetic vector potential $A_z(\rho, \phi, z)$ in a cylindrical coordinate system. A_z is related to the fields in the cavity by [6]

$$E_\rho = -j\frac{1}{\omega\mu\varepsilon}\frac{\partial^2}{\partial\rho\partial z}A_z \qquad H_\rho = \frac{1}{\mu}\frac{1}{\rho}\frac{\partial}{\partial\phi}A_z \tag{2.22a}$$

$$E_\phi = -j\frac{1}{\omega\mu\varepsilon}\frac{1}{\rho}\frac{\partial^2}{\partial\phi\partial z}A_z \qquad H_\phi = -\frac{1}{\mu}\frac{\partial}{\partial\rho}A_z \tag{2.22b}$$

$$E_z = -j\frac{1}{\omega\mu\varepsilon}\left(\frac{\partial^2}{\partial z^2} + k^2\right)A_z \qquad H_z = 0 \tag{2.22c}$$

By applying the boundary conditions of the PEC walls (zero tangential electric field $\mathbf{E}_t = E_\rho$ at the top and bottom surfaces $z' = h$ and $z' = 0$) and PMC walls (zero tangential magnetic field $\mathbf{H}_t = H_\phi$ at the cylindrical outer wall $\rho' = r$) to (2.22) and applying (2.22) to the general solution of (2.12), the solution reduces to [6]

$$A_z = B_{mnp}J_m(k_\rho\rho')[A_2\cos(m\phi') + B_2\sin(m\phi')]\cos(k_z z') \tag{2.23}$$

where

$$k_\rho^2 + k_z^2 = \omega_r^2\mu\varepsilon \tag{2.24}$$

and (ρ', ϕ', z') is a point within the cavity. $J_m(x)$ is an mth order Bessel function of the first kind, and

$$k_\rho = \frac{\chi'_{mn}}{r} \qquad m = 0, 1, 2, \ldots \quad n = 0, 1, 2, \ldots \tag{2.25}$$

$$k_z = \frac{p\pi}{h} \qquad p = 0, 1, 2, \ldots \tag{2.26}$$

where χ'_{mn} represents the mnth zero of the Bessel function. The lowest zero of $J_m(x)$ is $\chi'_{11} \approx 1.8412$, so the operating frequency of the lowest mode of the circular patch can be calculated as

$$f_{r,110} = \frac{1}{2\pi\sqrt{\mu\varepsilon}}\left(\frac{\chi'_{mn}}{r}\right) = \frac{1.8412}{2\pi r\sqrt{\mu\varepsilon}} \tag{2.27}$$

A similar treatment to that in the previous section can be used to transform the fields of the cavity model of the circular patch into a magnetic surface current $\mathbf{M}_s$ tangential to the cylindrical wall of the cavity and running parallel to ϕ, which can be used to calculate the far-field electric fields.

2.2 Reconfigurable Patch Antennas

As mentioned in section 2.1, microstrip patch antennas are readily amenable to the addition of mechanisms which enable dynamic reconfiguration of their operating frequency, polarization, and/or radiation pattern. Numerous different mechanisms have been explored to enable this reconfiguration. The bulk of these approaches take one of two approaches to reconfiguration:

1. *Switching mechanisms*, which effect change in the electrically active geometry of the antenna structure

2. *Loading mechanisms*, which apply a variable reactive load to the fields in the antenna structure

The reconfiguration mechanisms explored in this work can be broadly classified into two types: *electronic* mechanisms and *fluidic* mechanisms.

2.2.1 Electronic Reconfiguration Mechanisms

A wide variety of electronic mechanisms have been explored in literature to achieve reconfigurability in patch antenna designs. Several of the most common electronic reconfiguration mechanisms are PIN diodes [9–13], varactor diodes [14–17], and RF MEMS (micro-electromechanical systems) [18–21]. Of these three approaches, PIN diodes fall into the class of *switching mechanisms* whereas varactor diodes are most frequently used as a *loading mechanism*. RF MEMS can be used in either strategy, although they are most commonly used as a switching mechanism.

Although PIN diode-based reconfiguration is the only approach explored in this work, a brief overview of varactor & RF MEMS follows. Varactor diodes (also known as varicaps) are electronic devices that operate as voltage-controlled capacitors. The construction of a varactor is fundamentally the same as that of a conventional diode. They consist of a semiconductor p-n junction wherein two types of semiconductor with doping such that one region's majority charge carrier is positive (p-type) and the other region's majority charge carrier is negative (n-type) are joined together. In contrast to conventional diodes, however, varactors are almost universally operated in reverse-bias conditions, where the cathode (n-type semiconductor) is at a positive voltage potential relative to the anode (p-type semiconductor). In these conditions, the applied electric field forces the charge carriers in the p- and n-regions to separate from one another, generating a *depletion region* at the p-n boundary with relatively few charge carriers (and thus relatively good insulating properties). By varying the applied reverse voltage the width of the depletion region can be varied, effectively forming a parallel-plate capacitor with a voltage-variable plate distance. In operation, the capacitance of the varactor is inversely proportional to the applied DC bias voltage. In reconfigurable antenna systems, this variable capacitance is typically used as a variable reactive load on the antenna structure, or as a variable electrical length element in such a structure.

RF MEMS are miniature or microscopic systems, typically fabricated using planar semiconductor manufacturing techniques and often using similar materials, which utilize electrostatic forces (with electrostatic attraction being most common) to actuate micromachined structures to accomplish ohmic or capacitive switching, or to vary the distance between two conductive microstructures to vary the capacitance between them. Compared to true solid-state approaches such as PIN or varactor diodes, RF MEMS require significantly higher voltages to actuate—up to several hundred volts—although at extremely low current.

PIN diodes are constructed in a similar manner to conventional or varactor diodes, but with one key difference. During the doping process in which the semiconductor is formed into regions of p-type and n-type, a layer of *intrinsic*, or undoped, semiconductor is left separating the p- and n-regions. This intrinsic region has relatively few unbound charge carriers, so it presents a high resistance to the flow of current in an unbiased state. The relatively wide intrinsic region makes the resulting diode a poor rectifier at low frequencies, but at RF and microwave frequencies, it behaves as a current-variable resistor. The RF resistance of a PIN diode is inversely proportional to the DC bias current applied, and can vary from $10k\Omega$ at zero bias current down to as little as 0.1Ω with bias currents on the order of 1–10mA. Thus, PIN diodes see the most frequent use—including in this work—as a voltage/current controlled switch.

2.2.2 Fluidic Reconfiguration Mechanisms

While electronic antenna reconfiguration techniques have seen quite a bit of research, a relatively new class of *fluidic* antenna reconfiguration techniques have received a good bit of attention in recent years. Fluidic systems can be used in a variety of ways to reconfigure the operating behavior of an antenna, ranging from loading mechanisms [22–25] that use fluidic systems to vary a reactive or dielectric load on a reconfigurable antenna to geometry-manipulation mechanisms [26–31], which utilize fluidic systems to manipulate the physical geometry of the antenna.

The fluid systems explored in literature for reconfiguring the operating behavior of antenna systems can be broadly classified into two groups by the type of fluid utilized. They are

1. *Conductive* fluidic systems, which use conductive fluids (frequently liquid metals)

2. *Dielectric* fluidic systems, which use non-conducting fluids as the reconfiguration medium

In this work, both types of fluidic reconfiguration mechanism are explored. The following presents an overview of these fluid mechanisms.

2.2.2.1 Liquid Metal

Liquid metals are a collection of materials which exhibit the properties of metals—namely high electrical & thermal conductivity—and remain liquid at or below room temperature (20–25°C). Only one elemental metal falls into this category: Mercury. Several metal alloys, however, have melting points at or below room temperature. Principally, these are alloys containing either Gallium or Sodium. Table 2.1 shows a summary of the compositions and melting points of these liquid metals.

Table 2.1: Summary of liquid metal properties

Name	Chemical Composition (wt %)	Melting Point
Mercury	Mercury: 100%	-38.8°C
NaK	Sodium: 23% Potassium: 77%	-12.6°C
eGaIn	Gallium: 75% Indium: 25%	15.5°C
Galinstan	Gallium: 68% Indium: 22% Tin: 10%	11°C

Of these liquid metals, eGaIn (eutectic Gallium-Indium alloy) and Galinstan (Gallium-Indium-Tin alloy) are relatively inert. Mercury possesses the lowest melting point of the liquid metals, but it is highly toxic and exhibits a relatively high vapor pressure, meaning it evaporates readily and poses an inhalation toxicity risk to personnel. NaK (Sodium-Potassium alloy) also has a relatively low melting point, but it is also highly reactive. NaK reacts violently with water to form sodium and potassium hydroxides, hydrogen gas, and copious amounts of heat. It also reacts with air to form potassium oxides and superoxides, including the potent oxidizer KO_2, which can form a shock-sensitive explosive mixture with many organic compounds. Thus, both Mercury and NaK pose significant risks which greatly outweigh their potential benefits in antenna reconfiguration applications. Of these liquid metals, only eGaIn was explored in this work.

The basic principle of liquid metal reconfiguration mechanisms is to change the conductor geometry of the antenna structure to achieve reconfiguration of the antenna's operating behavior. Frequently, this geometry change is accomplished via pressure-driven displacement of the liquid metal [27–30], although the liquid metal can also be used as a highly flexible conductor in a flexible support, which can be reshaped mechanically using some external influence [31]. In this work, a pressure-driven liquid metal fluid network is explored.

2.2.2.2 Dielectric Fluids

Dielectric fluids are substances that are fundamentally non-conductive. A wide variety of dielectric fluids have been explored in antenna reconfiguration applications, [22, 24–27, 32, 33] with techniques ranging from variable dielectric loading for frequency reconfiguration to variable dielectric coupling for polarization reconfiguration. Table 2.2 gives dielectric properties for several such common fluids.

Table 2.2: Dielectric properties of selected dielectric fluids

Material	Relative Permittivity ε_r	Loss Tangent $\tan\delta$	at Frequency	Reference
Silicone Oil	2.74	0.1	3 GHz	[34]
Fluorinert FC-70	1.98 [35]	0.0013 [36]	213 GHz [36]	[35]
Hydrocal 2400	2.2	0.0008	10 GHz	[34]
Deionized Water	80	0.12	2.45 GHz	[37]
Methanol	31.7	0.289	1 GHz	[38]
Acetone	21	0.054	2.45 GHz	[37]

All dielectric materials can be fully characterized by their complex absolute permittivity

$$\varepsilon^* = \varepsilon' - j\varepsilon'' \tag{2.28}$$

where ε' is the real component and ε'' is the imaginary component. For most dielectric materials, ε^* varies as a function of frequency. Often, the dielectric properties of a material are specified by their *relative* permittivity ε_r and *loss tangent* $\tan\delta$ (as in Table 2.2). These terms are related to the complex permittivity ε^* by

$$\varepsilon_r = \frac{|\varepsilon^*|}{\varepsilon_0} \quad \text{and} \quad \tan\delta = \frac{\varepsilon''}{\varepsilon'} \tag{2.29}$$

For antenna reconfiguration applications it is key that a dielectric fluid have as low a $\tan\delta$ as practicable. For both the variable loading and variable coupling reconfiguration techniques, if the dielectric fluid has a high loss tangent the radiation efficiency of the antenna will be adversely impacted as electromagnetic energy in the antenna structure will be absorbed by the dielectric fluid and converted to waste heat instead of being radiated into free space.

Beyond pure, single-component dielectric fluids, another interesting class of dielectric fluids are *electromagnetically-functionalized colloidal dispersions* (EFCDs). EFCDs are multi-component dielectric fluids comprised of a continuous-phase dielectric fluid with a dispersed colloidal dielectric material, typically in nanoparticle form. A prime example of an EFCD is a dispersion of colloidal barium strontium titanate (BSTO) in a hydrotreated napthenic mineral oil such as Hydrocal 2400. [23] The key feature of an EFCD that makes it an attractive for antenna reconfiguration applications is it provides a mechanism to smoothly vary ε_r for the bulk EFCD over a wide range of values. By varying the volume fraction of colloidal BSTO (which exhibits a very large relative permittivity: $\varepsilon_r \approx 200$—$1000$) dispersed in the continuous-phase liquid, the overall ε_r of the EFCD can be varied from that of the continuous-phase liquid to a relatively high value (one reported range is $\varepsilon_r = 2.1$—8.3 [23]). Thus, an EFCD composed of Fluorinert FC-70 fluorocarbon oil and colloidal BSTO nanoparticles is considered in this work as the basis of a dielectric fluid reconfiguration mechanism.

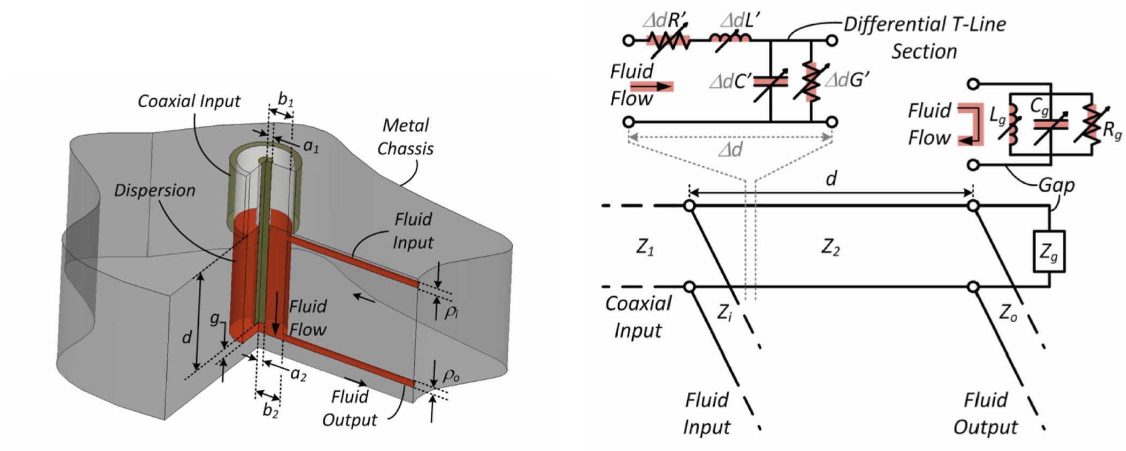
2.2.2.3 Coaxial Stub Microfluidic Impedance Transformers (COSMIX)

The coaxial stub microfluidic impedance transformer (COSMIX) was first presented in [23] as a readily adaptable impedance tuning mechanism to exploit a tunable EFCD to act as a potentially low-loss, widely variable reactive load for tunable RF structures. The geometry of the COSMIX is shown in Fig. 2.5a. The COSMIX geometry, as the name implies, is derived from a terminated length of coaxial transmission line. The center conductor of the line is separated from the bottom termination of the coaxial stub by a gap of width g. This geometry makes the COSMIX behave electrically as a finite length transmission line terminated by a capacitor, as shown in Fig. 2.5b. The theory of operation of the COSMIX centers around the flow of dielectric fluid in hollow cylindrical region between the inner and outer conductors. By controlling the relative permittivity of an EFCD pumped through the dielectric space, the COSMIX can be made to behave akin to a variable-length transmission line terminated by a variable capacitor. With the proper COSMIX length & width and a sufficiently wide range of permittivity in the EFCD, the impedance as seen at the coaxial input port can be varied to any reactive load.

In the fluidic reconfigurable antenna design developed in this work, COSMIX elements are explored as a reactive loading mechanism to achieve impedance bandwidth & operating frequency tuning. To achieve system-level control of the COSMIX elements, a preliminary system to vary the permittivity of an EFCD pumped through the COSMIX elements is proposed.

2.3 Phased Antenna Array Control

Phased antenna arrays are a popular and widely applicable method for constructing a high gain, electronically-steerable antenna system. The basic principle



(a) Physical geometry & design variables (b) Transmission line model

Fig. 2.5: Geometry & behavior of the COSMIX, Reprinted with permission from [23], © 2010 IEEE

underlying the operation of a (linear) phased array is as follows: in operation, the relative phase of the excitation of each antenna element in the array is controlled. In the far-field region (commonly defined as the region $2D^2/\lambda$ away from the array, where D is the largest dimension of the array and λ is the wavelength of the operating frequency), the fields emitted by each antenna element constructively interfere at the beam steering angle θ_0, producing a maximum lobe in the array's radiation pattern at θ_0. By changing the relative phase of each element's excitation, the angle at which this constructive interference occurs can be steered, effectively steering the maximum lobe of the array's radiation pattern.

When considering a real antenna element with a non-uniform radiation pattern, the overall radiation pattern of the array can be expressed as [1]

$$E_{\text{total}}(\theta, \phi) = E_{\text{element}}(\theta, \phi) \times \text{AF}(\theta, \phi) \tag{2.30}$$

where $E_{\text{total}}(\theta, \phi)$ is the far-field pattern of the array, $E_{\text{element}}(\theta, \phi)$ is the far-field pattern of each element in the array, and $\mathrm{AF}(\theta, \phi)$ is the *array factor*. For a uniform linear array with equal amplitude excitations, the array factor can be expressed as

$$\mathrm{AF} = \sum e^{j(n-1)\psi} \qquad \text{where} \quad \psi = kd\cos\theta + \beta \tag{2.31}$$

where $k = 2\pi/\lambda$, d is the spacing between elements, θ is the scan angle, and β is the progressive phase shift of the excitation between adjacent elements in the array.

(2.31) can be further reduced and normalized to [1]

$$\mathrm{AF}_n = \frac{1}{N}\frac{\sin\left(\frac{N}{2}\psi\right)}{\sin\left(\frac{1}{2}\psi\right)} \tag{2.32}$$

where N is the number of elements in the array. The concept of pattern multiplication as delineated in (2.30) can be used to extend the preceding discussion of 1D linear arrays to apply to a 2D planar antenna array, as well. For a rectangular planar array oriented along the x-y-plane, the array factor can be expressed as the product of two linear array factors in the x- and y-axes, respectively. Thus, for the planar rectangular array we have

$$\mathrm{AF}_n(\theta, \phi) = \frac{1}{M}\frac{\sin\left(\frac{M}{2}\psi_x\right)}{\sin\left(\frac{1}{2}\psi_x\right)} \times \frac{1}{N}\frac{\sin\left(\frac{N}{2}\psi_y\right)}{\sin\left(\frac{1}{2}\psi_y\right)} \tag{2.33}$$

where

$$\psi_x = kd_x \sin\theta\cos\phi + \beta_x \qquad \text{and} \qquad \psi_y = kd_y \sin\theta\sin\phi + \beta_y \tag{2.34}$$

and d_x is the element spacing along the x-axis, d_y is the spacing along the y-axis

In order to steer the beam to a desired (θ_0, ϕ_0), (2.34) can be solved for β_x and β_y, the progressive phase shifts along the x- and y-axes to find the beam steering

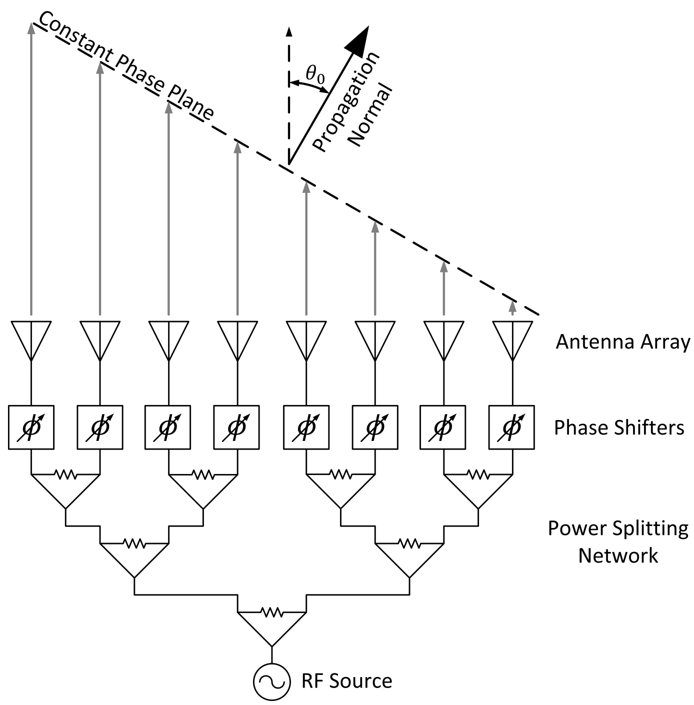


Fig. 2.6: Uniform linear phased antenna array with corporate feed network from common RF source

equations:

$$\beta_x = -kd_x \sin\theta_0 \cos\phi_0 \tag{2.35a}$$

$$\beta_y = -kd_y \sin\theta_0 \sin\phi_0 \tag{2.35b}$$

(2.35) can also be used to find the phase shifts for a planar array with non-rectangular element spacing, too. By referencing every element in the array to a common origin point, each element's (x, y) coordinates can be substituted into d_x and d_y to find the phase shifts along the x- and y-axes. Each element's individual

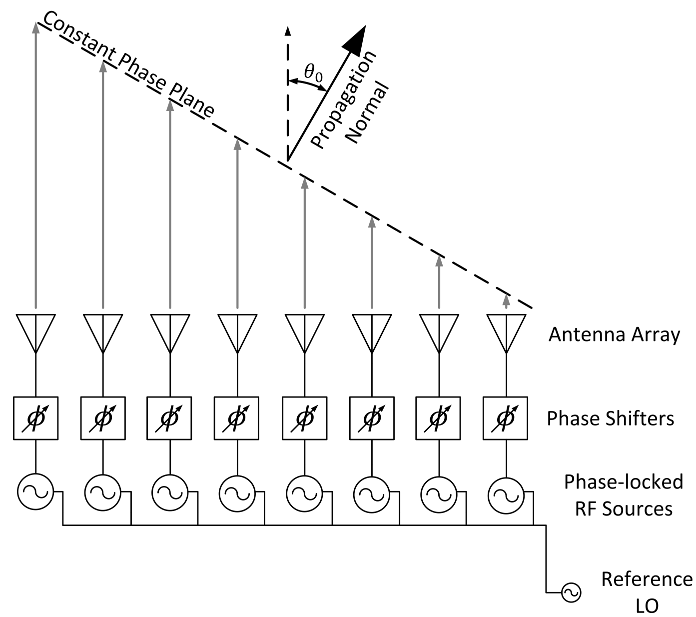


Fig. 2.7: Uniform linear phased antenna array with multiple phase-locked transmitters

excitation phase can then be computed as

$$\phi_e = -kx' \sin\theta_0 \cos\phi_0 - ky' \sin\theta_0 \sin\phi_0 \tag{2.36}$$

where (x', y') are the locations of the element centers relative to the origin of the array. In practice—including in this work—it is customary to normalize the phase shifts computed from (2.36) to either the most positive or most negative phase, and then compute the phases of the other elements relative to this most advanced or most retarded phase element. Furthermore, when the excitation signal being steered is narrowband, the computed phase shifts can be further computed as modulo 2π, as a narrowband signal roughly approximates a pure sinusoid, which is invariant in a full 2π phase shift.

Fig. 2.6 illustrates an example of a corporate-fed uniform linear array. The corporate feed network consists of a single RF source, the output of which is fed through an equiphase, equal-power splitting network. The outputs of this splitting network then pass through a set of variable phase elements before being fed into the antenna elements. This configuration—also known as a passive electronically-scanned array (PESA)—is the feed configuration explored in this work.

An alternate approach to feeding a phased array is the active electronically-scanned array (AESA) shown in Fig. 2.7. In this configuration, each array element has its own RF source locked to a common reference local oscillator, typically with a phase-locked loop (PLL) circuit. In AESA topologies, the phase shifting element can be placed either between the RF source and antenna as shown in Fig. 2.7 or it can be located between the RF source and the reference LO.

2.4 Direction of Arrival Estimation

Fundamentally, direction of arrival (DoA) estimation is the inverse problem of phased array control. Real-time DoA estimation with an antenna array requires a set of phase-sensitive receivers for every antenna element in the array, as shown in Fig. 2.8. By measuring the relative phase of the signal received at each antenna from an emitter in the far-field, the direction from which the signal arrived at the array can be computed.

The naïve approach to the DoA problem is to attempt to directly invert calculation process used to beamsteer the phased array. Recently, however, a variety of estimation algorithms have been developed & explored to not only speed up the DoA estimation process by reducing computational complexity but also to increase the spatial resolution beyond that achievable by conventional means. Of the variety of DoA estimation algorithms explored in literature, the Multiple Signal Classification (MUSIC) algorithm is one of the most popular and is the method explored in

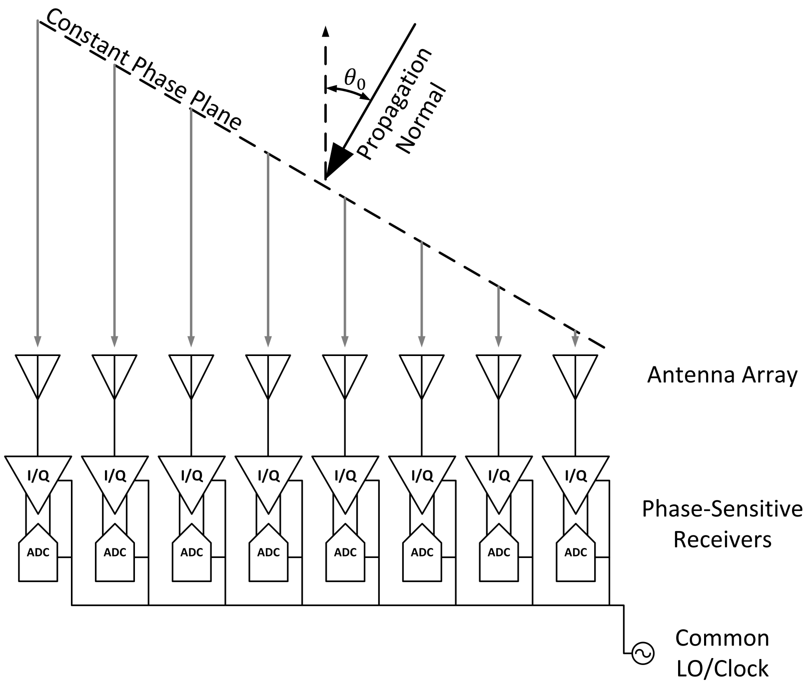


Fig. 2.8: Direction of arrival estimation using multiple phase-locked receivers

this work.

2.4.1 MUSIC Algorithm

MUSIC belongs to the family of DoA estimation algorithms known as *subspace* methods. MUSIC is often referred to as a type of *superresolution* DoA estimation algorithm, as it allows a much finer resolution of closely spaced emitters than conventional inverse-beamforming-type methods. The MUSIC algorithm proceeds as follows:

Given an array of M elements with M complex received signal weights (vectors of the form Ae^{jB}) and a set of D signals impinging on the array from D different directions, we can define an MxM *array correlation matrix* R_{xx} where each element R_{ij} is the product of the received signal weights x_i of element i and x_j of element j

such that [39]

$$
\begin{aligned}
R_{xx} &= E\left[\bar{x}\bar{x}^H\right] && \text{(2.37a)}\\
&= E\left[\left(\bar{A}\bar{s}+\bar{n}\right)\left(\bar{s}^H\bar{A}^H+\bar{n}^H\right)\right] && \text{(2.37b)}\\
&= \bar{A}E\left[\bar{s}\bar{s}^H\right]\bar{A}^H + E\left[\bar{n}\bar{n}^H\right] && \text{(2.37c)}\\
&= \bar{A}R_{ss}\bar{A}^H + R_{nn} && \text{(2.37d)}
\end{aligned}
$$

where R_{ss} is the DxD *source correlation matrix* among the D signal sources and $R_{nn} = \sigma^2 I$ is the MxM *noise correlation matrix* with random variance σ. By assuming that the noise in R_{nn} is uncorrelated with the signals in R_{ss}, we can assume that R_{xx} is Hermitian with M eigenvalues $(\lambda_1, \lambda_2, \ldots, \lambda_M)$ and M associated eigenvectors $\bar{E} = [\bar{e}_1\bar{e}_2\ldots\bar{e}_M]$. Sorting the eigenvalues in descending order enables the eigenvector matrix $\bar{E}$ to be partitioned into a D-vector signal subspace and a $M-D$-vector noise subspace, or $\bar{E} = \left[\bar{E}_N\bar{E}_S\right]$.

Finally, the eigenvectors in $\bar{E_N}$ are assumed to be orthogonal to the steering vectors of the array elements $a_M(\theta,\phi)$ at the angle of arrival (θ_0,ϕ_0). Based on this assumption, the Euclidean distance between the noise subspace eigenvectors and the array steering vectors from the received signal phases can be assumed to be roughly zero at the angle of arrival. This distance is calculated: $d^2 = \bar{a}(\theta',\phi')^H\bar{E}_N\bar{E}_N^H\bar{a}(\theta',\phi')$ Because this distance will be minimized when the search angles (θ',ϕ') coincide with the angle of arrival (θ_0,ϕ_0), this expression is placed in the denominator of a pseudospectrum function whose peaks then correspond to the directions of arrival for the D signals:

$$
P_{\text{MUSIC}} = \frac{1}{\bar{a}(\theta',\phi')^H\bar{E}_N\bar{E}_N^H\bar{a}(\theta',\phi')} \tag{2.38}
$$

Thus, by finding the maximum of this pseudospectrum function in the search space, the DoA of an incident signal can be estimated. [40]

2.5 Thermoelectric Cooling

Thermoelectric cooling is a solid-state process in which electrical energy is used to directly move heat energy from one location to another. Thermoelectic effects occur when two dissimilar conductor or semiconductor materials are joined, and arise primarily from the difference in the band gap energy of the two materials. Thermoelectric effects fall into three categories of separately-discovered effects:

- *Seebeck effect*: The generation of an electrical potential and/or current from an imposed heat flux on a thermoelectric junction

- *Peltier effect*: The generation of a heat flux across a thermoelectric junction from an imposed electrical current

- *Thomson effect*: The generation of a heat flux across a single current-carrying conductor as a result of a temperature difference across the conductor

The Peltier and Seebeck effects are essentially the opposite effect of one another, and the Thomson effect is a continuous version of the Peltier effect that arises as a result of the temperature dependence of the Seebeck coefficient of many materials. It is the Peltier effect that is of particular interest in this work, as the Peltier effect allows the use of an electric current to directly generate a heat flux.

The result of the Peltier and Seebeck effects can be expressed as a relationship between the voltage, current, heat flux, and temperature difference across a thermo-

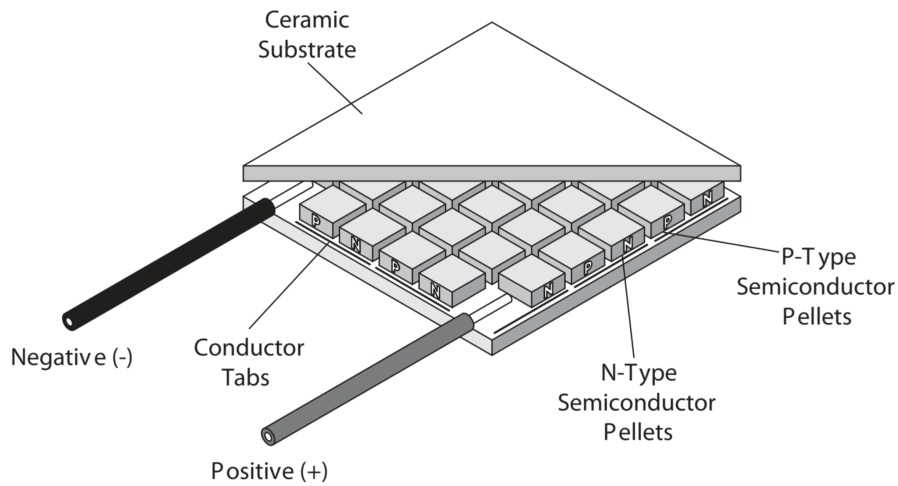


Fig. 2.9: Construction of a peltier thermoelectric cooling device, Reprinted with permission from [41], © 2009 CUI, Inc.

electric junction, shown in (2.39).

$$\begin{pmatrix} V \\ \dot{Q} \end{pmatrix} = \begin{pmatrix} R & S_{AB} \\ \Pi_{AB} & -\kappa \end{pmatrix} \begin{pmatrix} I_{el} \\ \Delta T \end{pmatrix} \tag{2.39}$$

In this expression, the total voltage is the sum of the applied voltage V_{el} and thermally-induced voltage V_{th}, $V = V_{el} + V_{th}$ and likewise the total current $I = I_{el} + I_{th}$. R is the electrical resistance of the junction, $S_{AB} = S_A - S_B$ is the difference in the Seebeck coefficient between materials A and B in the junction, and $\Pi_{AB} = \Pi_A - \Pi_B$ is the difference between the Peltier coefficients of the two materials. $\dot{Q}$ is the heat flux through the junction, κ is the thermal conductance of the junction, and ΔT is the temperature difference across the junction. Note that this expression is valid only in the linear regime of the junction, as R, S_{AB}, and Π_{AB} all vary with respect to ΔT. The terms in (2.39) are defined as follows:

$$S_{AB} = \lim_{\Delta T \to 0} -\frac{\dot{Q}}{\Delta T}\bigg|_{I=0} \tag{2.40}$$

$$\Pi_{AB} = \frac{\dot{Q}}{I_{el}}\bigg|_{\Delta T=0} \tag{2.41}$$

$$\kappa = \lim_{\Delta T \to 0} -\frac{\dot{Q}}{\Delta T}\bigg|_{I=0} \tag{2.42}$$

The thermally- and electrically-induced voltages and currents are further defined as:

$$V_{th} = S_{AB}\Delta T|_{I=0} \tag{2.43}$$

$$V_{el} = I_{el}R|_{\Delta T=0} \tag{2.44}$$

$$I_{th} = \frac{V_{th}}{R} = \frac{S_{AB}\Delta T}{R} \tag{2.45}$$

$$I_{el} = \frac{V_{el}}{R} \tag{2.46}$$

The key insight from (2.39) is that $\dot{Q} = \Pi_{AB}I_{el} - \kappa\Delta T$, so at $\Delta T = 0$ the heat flux through a Peltier junction is directly proportional to the electric current flowing through it (ignoring second order effects such as the ΔT dependence of Π_{AB}). Further, the sign of $\dot{Q}$ can be flipped by flipping the sign of I_{el}, so the direction of the applied electric current determines the direction of heat flow through the junction. Thus, a Peltier junction can be used as a reversible-direction, relatively-linear heat pump. Also of note from this expression is that the total pumped heat is inversely proportional to ΔT, due to Fourier's law of heat conduction: $\dot{Q} = -\kappa\Delta T$. Thus, as the temperature difference across the Peltier junction increases, the rate at which it pumps heat from the cold side to the hot side will decrease.

Fig. 2.9 shows a schematic of the construction of a commercial Peltier thermoelectric cooler similar to that used in this work. The module consists of a string

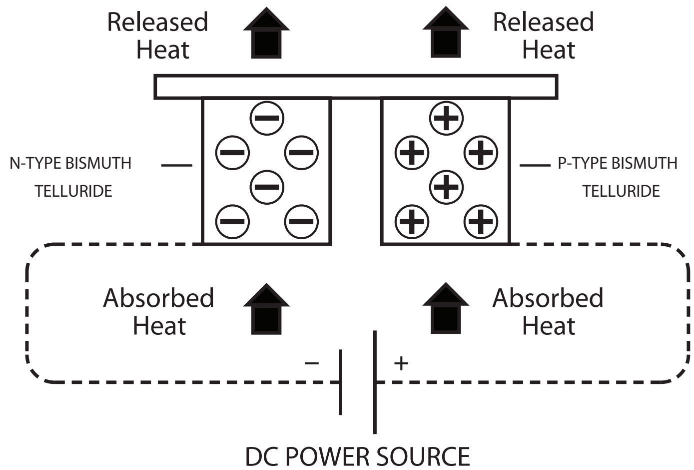


Fig. 2.10: Theory of operation of a Peltier TEC device, Reprinted with permission from [41], © 2009 CUI, Inc.

of Bismuth Telluride semiconductor pellets, alternately doped as n-type or p-type. The top and bottom faces of adjacent pellets are joined together by metal conductor tabs, such that the circuit through the module is a series connection of metal→n-type→metal→p-type. The pellets are arranged such that they are thermally in parallel, and bonded between two ceramic plates that provide high electrical and low thermal resistance.

Fig. 2.10 shows a graphical representation of the theory of operation of a Peltier thermoelectric cooling device. When the polarity of the DC power source is reversed in this figure, the flow of heat will reverse as well.

2.6 Proportional-Integral-Derivative Control

Proportional-Integral-Derivative—or PID—control is one of the oldest and most common control algorithms in use today. PID process control theory emerged out of mechanical governor design in the 1890s, and was first developed into a full theoretical model by Nicolas Minorsky in 1922 [42]. The theory was based on the observation

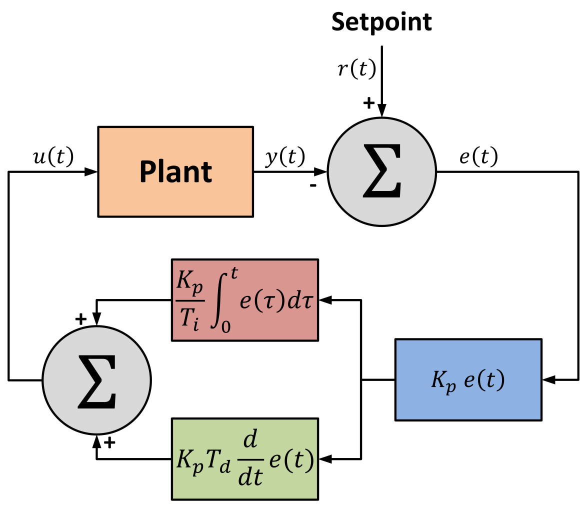


Fig. 2.11: Canonical interacting PID controller

of the helmsman of naval vessels, who based their steering inputs not only on the currently observed course error, but also on the historical (previous) course error and the current rate at which the error was changing (increasing or decreasing). This is the fundamental process by which all PID controllers operate, and is illustrated graphically by Fig. 2.11.

The key advantage of PID control is that its implementation does not require any mathematical model of the process to be controlled, which is a distinct advantage when used to control a complex or multi-part physical process. The key disadvantages of PID control are that it does not guarantee optimal control of a given system, requires a tuning procedure to derive the tuning parameters for the P, I, & D terms, and is fundamentally a linear control scheme so it can and does have difficulty controlling some non-linear processes. Despite these disadvantages, the relative simplicity of implementing PID control and the long history of study PID theory has undergone make it a robust and quite common choice for a control algorithm. In fact, it is estimated that roughly 95% of control loops implemented in the field of process control today are of the PID-type. [43] For this reason, PID control was chosen in the work as the closed-loop control algorithm for the implemented thermal control system.

The canonical PID controller functions as follows: First, the physical process (or *plant*) to be controlled is measured with some transducer instrument (a level gauge, temperature sensor, speed/position sensor, etc.), forming a *process value* measurement input, $u(t)$, to the controller. This value is subtracted from a *setpoint* value $r(t)$, which is the desired value of the measurement. This difference forms an error signal $e(t)$, which encapsulates the magnitude and direction of the measured process value's deviation from the setpoint. This error signal forms the input to the PID controller proper.

The output of the PID controller in Fig. 2.11 is expressed as

$$u(t) = K_p \left(e(t) + \frac{1}{T_i} \int_0^t e(\tau) d\tau + T_d \frac{d}{dt} e(t) \right) \tag{2.47}$$

In (2.47), K_p is the proportional gain, or *controller gain*, which affects the controller's response to error at time t. T_i is the integration time constant, which affects the controller's response to past error integrated from roughly time $t - T_i$ to time t. T_d is the derivative time constant, which gives the controller the ability to make a linear prediction of the error at time $t + T_d$. These terms are the tuning parameters for the controller, which allow the designer to tune the controller for a particular process to achieve a desired response time, overshoot, and settling time.

In practice, almost no PID control implementation takes the form of (2.47) directly. Nearly all modern PID controllers, including in this work, are implemented using a computerized system. This necessitates that the linear form of the PID controller be translated into a discrete form which can operate on a sampled version of the error signal. The discrete form of the interacting PID control algorithm at timestep n is shown in (2.48).

$$u(n) = K_p e(n) + K_i \sum_{k=0}^{n} e(k) + K_d \left(y(n) - y(n-1)\right) \tag{2.48}$$

This expression is of the discrete, *non-interacting* (or parallel) form, where

$$K_i = \frac{K_p T}{T_i} \quad K_d = \frac{K_p T_d}{T}$$

This expression incorporates the following discrete approximations:

$$\int_0^t e(\tau)d\tau \approx T \sum_{k=0}^{n} e(k) \tag{2.49}$$

$$\begin{aligned} \frac{d}{dt}e(t) &\approx \frac{1}{T}\left(e(n) - e(n-1)\right) \\ &\approx \frac{1}{T}\left(y(n) - y(n-1)\right) \end{aligned} \tag{2.50}$$

In (2.48)–(2.50), the algorithm is run at a sampling interval T, such that the current time can be expressed as $t = nT$ for some integer $n > 0$. Note also that the derivative of the error signal is approximated as the discrete difference of the process value, $y(n) - y(n-1)$. This is done so that a large change in the process setpoint $r(n)$ between times $(n-1)$ and (n) does not result in an erroneously large derivative term (commonly known as *bumpless* setpoint control). [44]

2.6.1 Integral Anti-Windup

Another key consideration in the implementation of a PID controller is that the control algorithms discussed above rely on assumptions of some degree of linearity in the process between the control output $u(t)$ and the error signal input $e(t)$. In a practical implementation, however, all physical processes are nonlinear to some degree. In particular, in any practical implementation the control signal $u(t)$ will be used to control some physical mechanism—a valve position, the flow of an electrical current, etc. All such physical mechanisms have some minimum and maximum limits: a valve cannot be opened or closed past its fully open or closed positions; the current is limited by the maximum current handling capability of the semiconductor devices and/or wiring used to control & transmit it.

When the output of a PID controller tries to drive its associated physical mechanism past its limits, the controller is said to have *saturated*. During saturation, the plant/process is no longer under closed-loop control, as the physical output has been decoupled from the controller's $u(t)$ output by the physical limiting mechanism. Controller saturation occurs primarily in two instances: during a large external disturbance to the physical process itself, and during a large change in the process setpoint. In the first case, if the disturbance large and continuous enough to hold the controller in saturation, then the process is said to have departed its controllable range. If the saturation condition is temporary, however, it can lead to a phenomenon known as *integral windup*. In the controller described by (2.48), a saturated output will result in the integral term accumulating a large value (either positive or negative) as it sums the large $e(n)$ during the saturation period. Once the process value approaches the setpoint and $|e(n)|$ begins to drop, the "wound-up" error sum in the integral term will drive the process value far past the setpoint until $e(n)$ is opposite-signed long enough to reduce the sum back down to the true closed-loop value.

Integral windup can thus lead to a variety of control problems, ranging from severe overshoot to complete process instability and oscillation. There have been explored a variety of approaches to combat the problem of integral windup, ranging from back-calculation to conditional integration [45], and combinations thereof. [46]

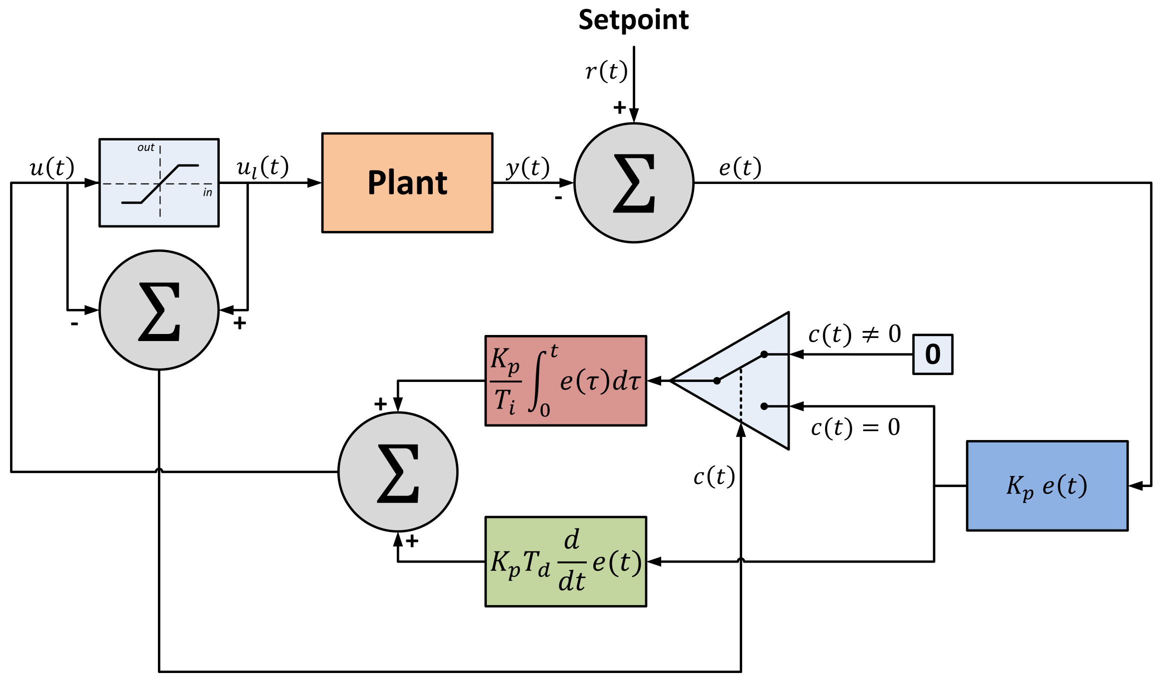


Fig. 2.12: Interacting PID controller with limited control output & conditional integration

In this work, conditional integration was chosen for its ease of implementation and its effectiveness in the implemented control system.

Conditional integration is implemented as shown in Fig. 2.12. In essence, the controller monitors both the calculated control output $u(t)$ and the actual, limited control $u_l(t)$ being applied to the process, calculating a conditional signal $c(t)$. When $u(t) = u_l(t)$, $c(t) = 0$ and the controller operates as normal. When the controller tries to drive the control mechanism beyond its physical limits such that $u(t) \neq u_l(t)$ the integrator is turned off (i.e. supplied with $e(t) = 0$). In this manner, the PID controller operates as a normal linear system during closed-loop control operation, but when a setpoint change or process disturbance drives the system into open-loop operation the erroneously large error signal is not integrated resulting in better, more predictable performance once the system re-enters closed-loop operation.

2.6.2 PID Tuning

As described above, the PID controller has a set of tunable parameters, K_p, T_i, and T_d, which control the relative effect of the proportional, integral, and derivative action respectively. Because the PID controller does not implement a mathematical model of the process under control, some method is required to determine the optimal or desired values for these tuning parameters when a PID controller is connected to a given process. There has been considerable study into various methods for deriving the tuning parameters for a given PID control loop. These methods can be broadly classified into two types: [47]

- *Closed loop* methods, which involve operation of the process under automatic control (i.e. with the PID controller active)
- *Open loop* methods, where the process is operated under manual control, typically with the PID controller's output $u(t)$ disconnected from the process

Most PID tuning methods involve some specification of the desired response of the system after tuning. For the most popular methods in use today, this is typically the *quarter-amplitude-decay*, or QAD response, which is characterized by a very quick process response to disturbances or setpoint changes (sub-process time constant) with some overshoot, where the process value exhibits a damped oscillation around its new value. The term "QAD" comes from the fact that the oscillation amplitude decays such that the second period's amplitude is 1/4 of the first.

In general, closed loop methods generally require that the tuning parameters of the controller are modified such that the process is brought close to instability, then the oscillatory response of the process value is measured after a disturbance is applied (either to the physical process or setpoint). In one of the most popular methods, the

Ziegler-Nichols method,the process is induced into a steady-state oscillation condition by increasing the controller gain K_p and applying a step change to the setpoint. When the process is oscillating at steady state, the critical controller gain $K_c u$ and period of the oscillation P_u are measured and used to compute the tuning parameters according to [47]

$$K_p = 0.6 K_c u \tag{2.51}$$

$$T_i = \frac{1}{2} P_u \tag{2.52}$$

$$T_d = \frac{1}{8} P_u \tag{2.53}$$

The popularity of the Ziegler-Nichols tuning method is likely attributable to the relative simplicity of the calculations required, but the method has quite a few drawbacks. The key issue with the Ziegler-Nichols method is that it requires that the process be made to oscillate. If the process is unstable by nature, such a procedure brings the process dangerously close to instability that can result in out-of-control oscillation and physical damage to plant equipment. Furthermore, some processes are inherently overdamped and cannot be induced to oscillate by controller action alone, rendering the Ziegler-Nichols method useless.

Open-loop tuning methods, on the other hand, generally center around a measurement of the process' transient response to a step input at the control mechanism. These methods are useful for overdamped processes, such as the thermal system in this work, which cannot be easily induced into oscillation. The Cohen-Coon tuning method is the tuning method selected for use in this work, as it is designed to produce a QAD response which results in a quick process response to setpoint changes—the primary process disturbance in the application studied. Cohen-Coon assumes a first-order plus deadtime process model. The tuning process for the Cohen-Coon method is as follows:

1. Bring process online in manual control mode (PID controller disconnected)
2. Allow process value to settle to a constant steady-state value, record steady-state PV
3. Apply a step change in the control output
4. Measure and record process value's response until a new steady-state value is achieved
5. Calculate process' dead time, t_d, time constant τ, and gain g_p
6. Calculate K_p, T_i, and T_d from the Cohen-Coon tuning rules

The process gain is calculated as the total change in the process value during the step test divided by the change in the control output, or

$$g_p = \frac{\Delta y}{\Delta u}$$

The process dead time t_d is the delay between a change in the control output (process "input") and process value (process "output"). It is calculated by extrapolating a linear curve fit tangent to the maximum slope (maximum rate of change) of the process value, then finding the time difference between the step change in the control output and the intersection of this curve fit and the starting steady-state process value. The time constant τ is the time difference between the end of the dead time t_d and the time the process value has changed $(1 - e^{-1})\Delta y$, or $0.632\Delta y$. Fig. 2.13 and 2.14 show the measured step response with the calculated process parameters annotated.

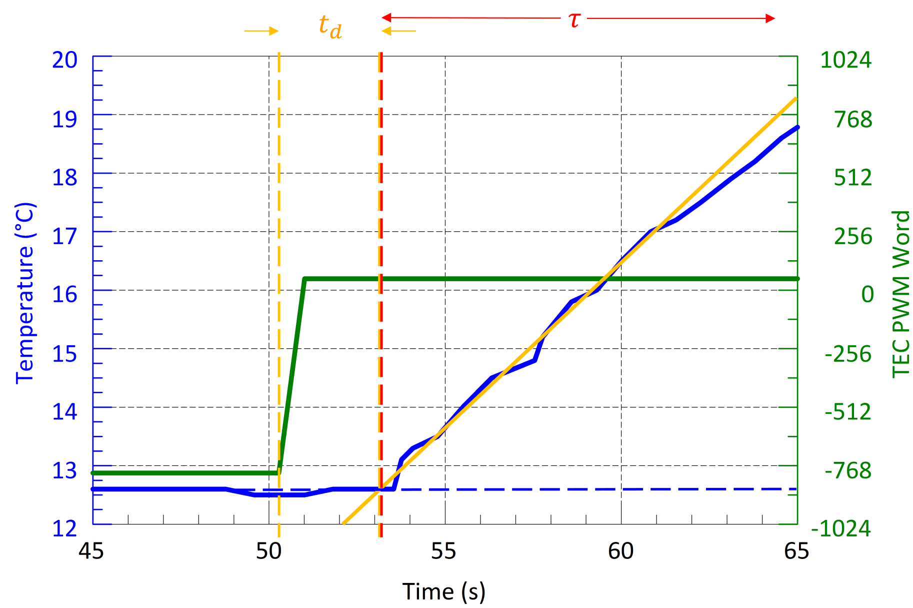


Fig. 2.13: Measured thermal control loop step response with process deadtime t_d annotated

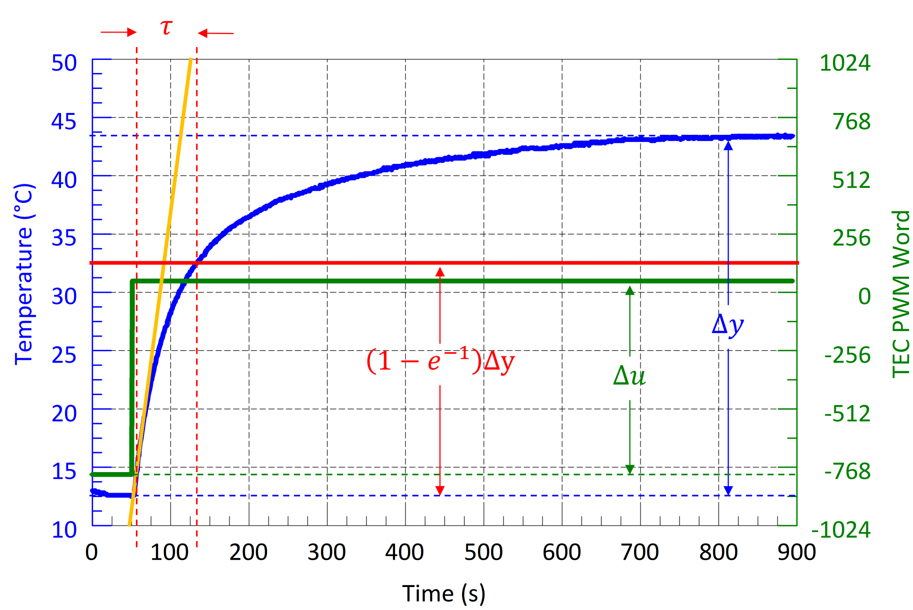


Fig. 2.14: Measured thermal control loop step response with time constant τ and gain $g_p = \Delta y/\Delta u$ annotated

Once the necessary step response characteristics have been calculated, the tuning parameters can be calculated according to [47]

$$K_p = \frac{1}{g_p}\left(\frac{3t_d + 16\tau}{12t_d}\right) \tag{2.54}$$

$$T_i = t_d\left(\frac{2(16\tau + 3t_d)}{13\tau + 8t_d}\right) \tag{2.55}$$

$$T_d = t_d\left(\frac{4\tau}{11\tau + 2t_d}\right) \tag{2.56}$$

2.7 TCP/IP Suite & Client-Server Architecture

In the full multifunctional antenna reconfiguration control system, a computer network is used to distribute reconfiguration commands and data. One of the most common networking models in use today is the *Internet Protocol suite*, which is the model and protocol set used by the global Internet to communicate data between individual computers, or *hosts*. This network model was chosen because of its ubiquity and the ease of software development it offers. The IP suite is often referred to as the *TCP/IP* suite, including in this document, as the Transmission Control Protocol (TCP) and Internet Protocol (IP) are the two most important and widely used protocols in the suite.

The IP suite consists of two parts: the model that defines the abstraction layers used in the suite, and the protocols used to implement the layers of abstraction.

2.7.1 TCP/IP Model

The TCP/IP model defines the abstraction layers used in the TCP/IP suite. The TCP/IP model consists of four layers, in order of decreasing abstraction:

- Application layer
- Transport layer

- Internet layer
- Link layer

Fig. 2.15 shows a graphical representation of a data flow through this model. The arrows indicate data flow from one application process to another. The application layer is the layer at which software applications on individual hosts send and receive data. In this work, the application layer is where the custom module communication protocol was implemented. The transport layer is where protocols such as TCP provide end-to-end data transport services to applications. The Internet layer is where the Internet Protocol provides host addressing and data transmission services from one host to another on a computer network. The link layer is where the physical link communication protocols and modulation schemes are used to transmit data over a physical medium such as Fast Ethernet over unshielded twisted pair or Wi-Fi over a 2.4GHz radio channel.

In this work, TCP is used for the transport layer protocol and IP for the Internet layer protocol. Both the 802.11g Wi-Fi and 100BASE-TX Fast Ethernet standards are used at the link layer.

2.7.2 Client-Server Model

In the context of computer networking, a *client-server* network architecture is one in which a set of programs (clients) communicate with and request service from a listening server. The client-server architecture is typically implemented with a *request-reply* messaging system, in which the server continuously listens for requests from clients and replies to those clients with the requested data. In practice, individual programs can act purely as clients, purely as servers, or—as is the case in this work—as both client and server.

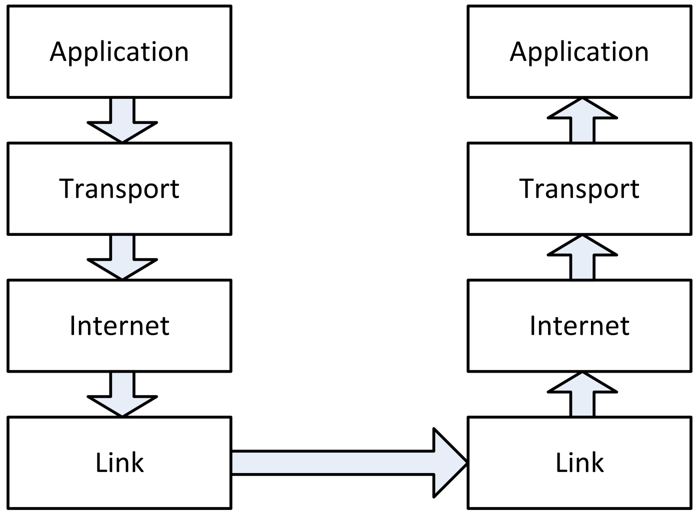


Fig. 2.15: TCP/IP layered data flow model

2.7.3 Transmission Control Protocol & Internet Sockets

The Transmission Control Protocol (TCP) is one of the most common transport layer protocols in use today. TCP provides reliable, ordered, and error-checked delivery of data between two applications. TCP connections between a client and server application are identified by a *source* and *destination* port number. Each port number is a 16 bit unsigned integer (0–65,535) reserved on the respective host by its respective application, and the combination of (source address, source port, destination address, destination port) serves to uniquely identify a *stream socket* which implements TCP to provide data communication between the client (sending) and server (receiving) applications.

TCP divides data passed from the application layer into *segments*, which are data structures consisting of a TCP header and a data payload. Each TCP segment header contains information about the source and destination ports for the data, a sequence number used by the receiver to ensure proper data order an acknowledgement number used in *ACK* (acknowledgement) segments to verify the proper reception of transmitted segments. The header also contains additional flag bits used during connection establishment and teardown, and a 16 bit checksum used to verify the integrity of the received segment.

TCP is a stateful protocol, meaning that both the client and server transition through a number of distinct states as a TCP socket is opened, used, and then closed. The *SYN*, or synchronize segment type is used to establish the TCP socket, the *ACK* segment type to acknowledge receipt of a previous segment, and the *FIN* segment type is used to tear down an established socket connection. In short, the the connection process proceeds as follows:

1. The server application opens a TCP socket on a port in the *listening* state and waits for an incoming *SYN* segment

2. The client reserves an *ephemeral port* for its source port, transmits a *SYN* to the server's port, and waits for an *ACK*

3. The server responds with a *SYN-ACK*, acknowledging the client's *SYN* and reciprocating the connection request, then waits for an *ACK*

4. The client *ACK*s the server's *SYN-ACK*, and both client and server transition into the *connection established* state

Once the connection is established, the the client and server then exchange data at the application layer:

1. The client application sends application request data segments, and waits for an *ACK*

2. The server *ACK*s the preceding application data

3. The server application replies with application response data segments, and waits for an *ACK*

4. The client *ACK*s the preceding application data

Once application data exchange is complete, either the client or server can initiate the teardown process to close the socket. When the client terminates, the sequence is:

1. The client sends a *FIN* segment and waits for an *ACK* and *FIN* from the server

2. The server *ACK*s the client's *FIN*, and sends its own *FIN*, then waits for an *ACK*

3. The client sends an *ACK*, closing its end of the socket

4. The server receives the client's *ACK* and the socket is fully closed

These processes are shown graphically in Fig. 2.16.

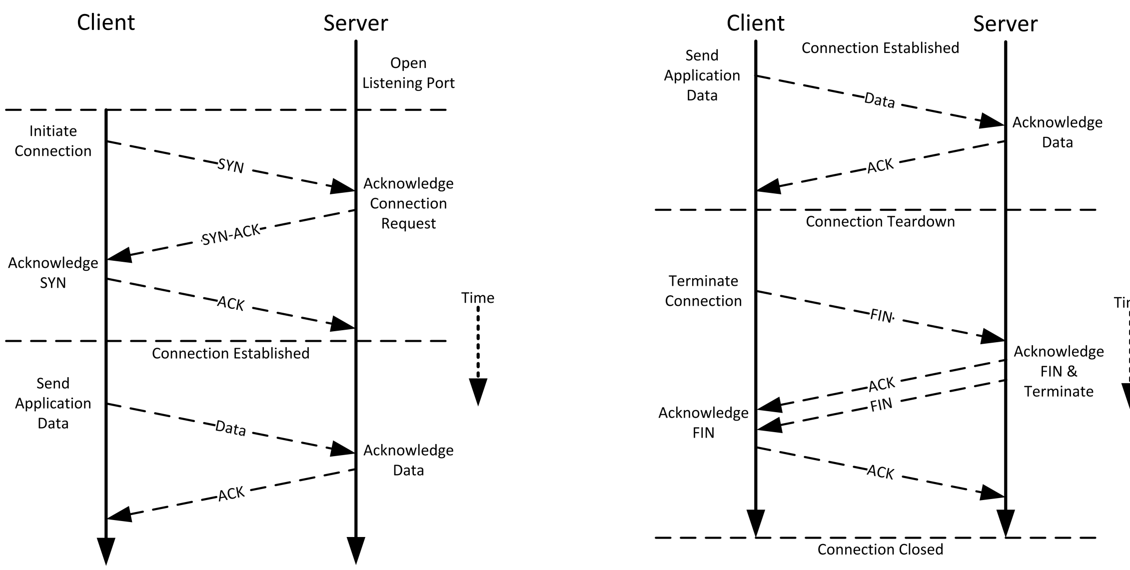


Fig. 2.16: TCP connection establishment process (right) and connection teardown process (left)

3. TRI-BAND POLARIZATION- & FREQUENCY-RECONFIGURABLE ANTENNA (TBPFRA)

3.1 Design Goals

The primary goal of the tri-band polarization- & frequency-reconfigurable antenna (TBPFRA) element design is to leverage fluidic reconfiguration techniques to achieve frequency and polarization agility in multiple bands over a wide frequency range. Simultaneously, the secondary goal of the antenna design is to maintain precise control over the physical antenna temperature to enable not only compensation for antenna element temperature rise due to losses during high RF power operation but also allow an array of such antenna elements to be used to send information by manipulating elements' temperature to create patterns in the array's thermal-infrared radiation. The frequency range of interest for this antenna design is L-band through mid-C-band (1–6 GHz). Thus, the design is targeted to have one operational mode each in the L-band, S-band, and C-band.

Table 3.1: IEEE standard letter designations for radar frequency bands [48]

Band Designation	Nominal Frequency Range
L-band	1–2 GHz
S-band	2–4 GHz
C-band	4–8 GHz

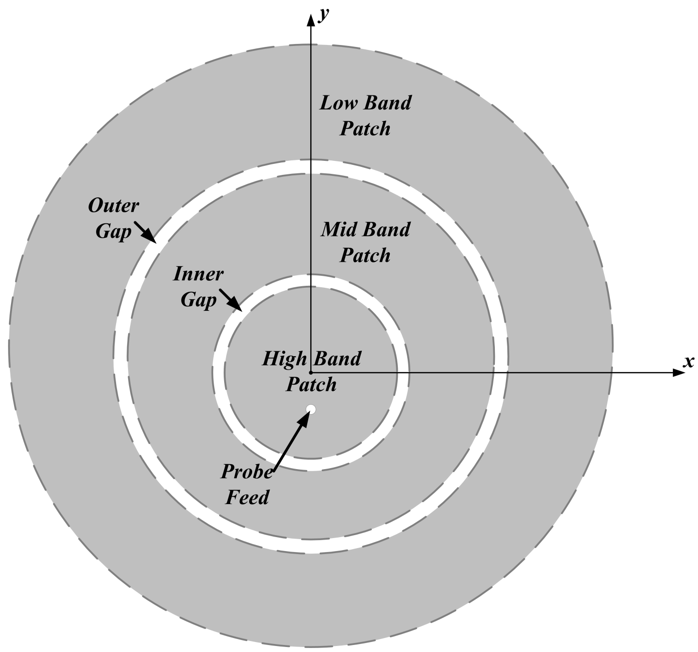


Fig. 3.1: Concentric circular patches analogous to TBPFRA

3.2 Design

3.2.1 Concept

The design concept for the geometry of the TBPFRA is based on a set of circular patch antennas, overlaid and sharing a common set of probe feeds. Fig. 3.1 shows a representative set of three circular patches, polarized along the y-axis, sharing a common probe feed. The patches are separated by two circular gaps. When both gaps are unoccupied only the innermost high-band patch is directly coupled to the probe feed, so the antenna operates at a frequency defined principally by the radius of the high-band patch (although capacitive coupling to the mid-band patch results in a lower operating frequency than would be achieved with a completely isolated high-band patch). When the inner gap is filled with a conductive fluid, the mid-band patch is directly coupled to the high-band patch and probe feed, so the antenna now

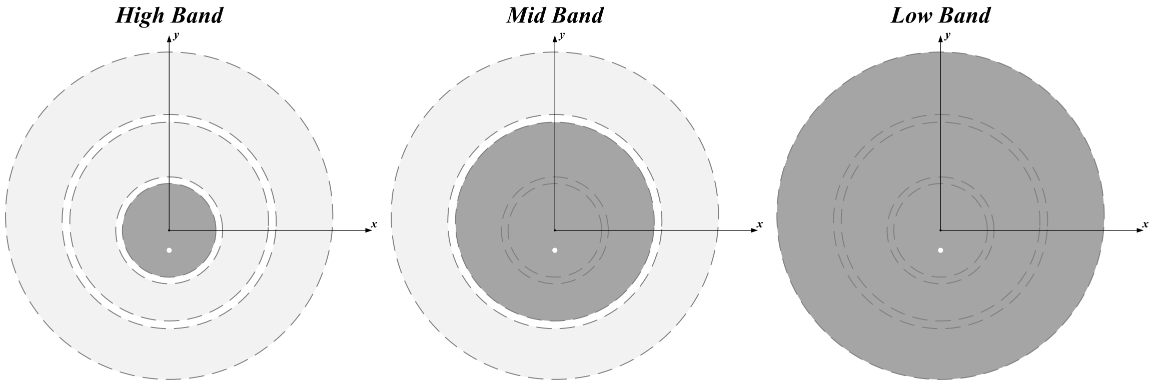


Fig. 3.2: Switching bands by filling gaps

operates at a lower frequency defined by the mid-band patch radius. Finally, when both the inner and outer gaps are filled with conductor, the antenna operates in its low-band mode. This concept is illustrated in Fig. 3.2. The design shown in Fig. 3.1 & 3.2 is capable of frequency reconfiguration, but all three frequency bands share the same polarization mode (linear polarization along the y-axis). In order to achieve polarization reconfigurability, this design is modified by adding an orthogonal probe feed on the x-axis, which will be excited independently of the y-axis feed. In order to achieve impedance matching at both probe feed locations, the patches are rotated 90° about the z-axis and sectioned to create two independent, orthogonal arms. This is the basis of the TBPFRA, shown in Fig. 3.3. This geometry allows both frequency and polarization reconfigurability. The dual feeds allow both the x-axis aligned and y-axis aligned polarization modes to be excited independently.

Fig. 3.4 shows the key design parameters for each of the two arms in the design. The high-band circular patch is centered on the origin. Three circles of radius a_l, a_m, and a_h define the outer extent of the low-band, mid-band, and high-band geometries respectively. These radii are used to set the operating frequency of the antenna in each of the three bands during tuning. An outer gap discontinuity of width g_o

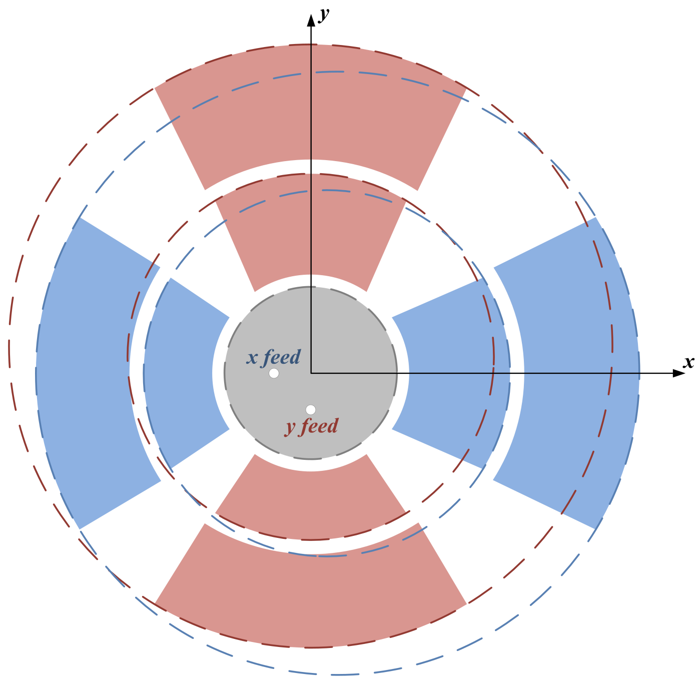


Fig. 3.3: TBPFRA metallization top view

separates the low-band and mid-band sectors. An inner gap discontinuity of width g_i separates the mid-band sectors from the high-band patch. These gaps electrically isolate the low- and mid-band sectors from each other and from the high-band patch. y_H defines the distance between the probe feed and the center of the high-band patch, and is used during tuning to impedance match the patch at its operating frequency. The distances y_M and y_L define the offsets between the center of the high-band patch and the centers of the mid-band arc circle and low-band arc circle, respectively. These parameters are used during tuning to impedance match the mid-band and low-band modes. $\theta_{M,i}$ and $\theta_{M,o}$ define the angles subtended by the inner and outer edges, respectively, of the two mid-band sectors. These two angles are referenced to the center of the circles defined by radii $a_h + g_i$ and a_m, respectively. Likewise, $\theta_{L,i}$ and $\theta_{L,o}$ define the angles subtended by the inner and outer edges of the two low-band

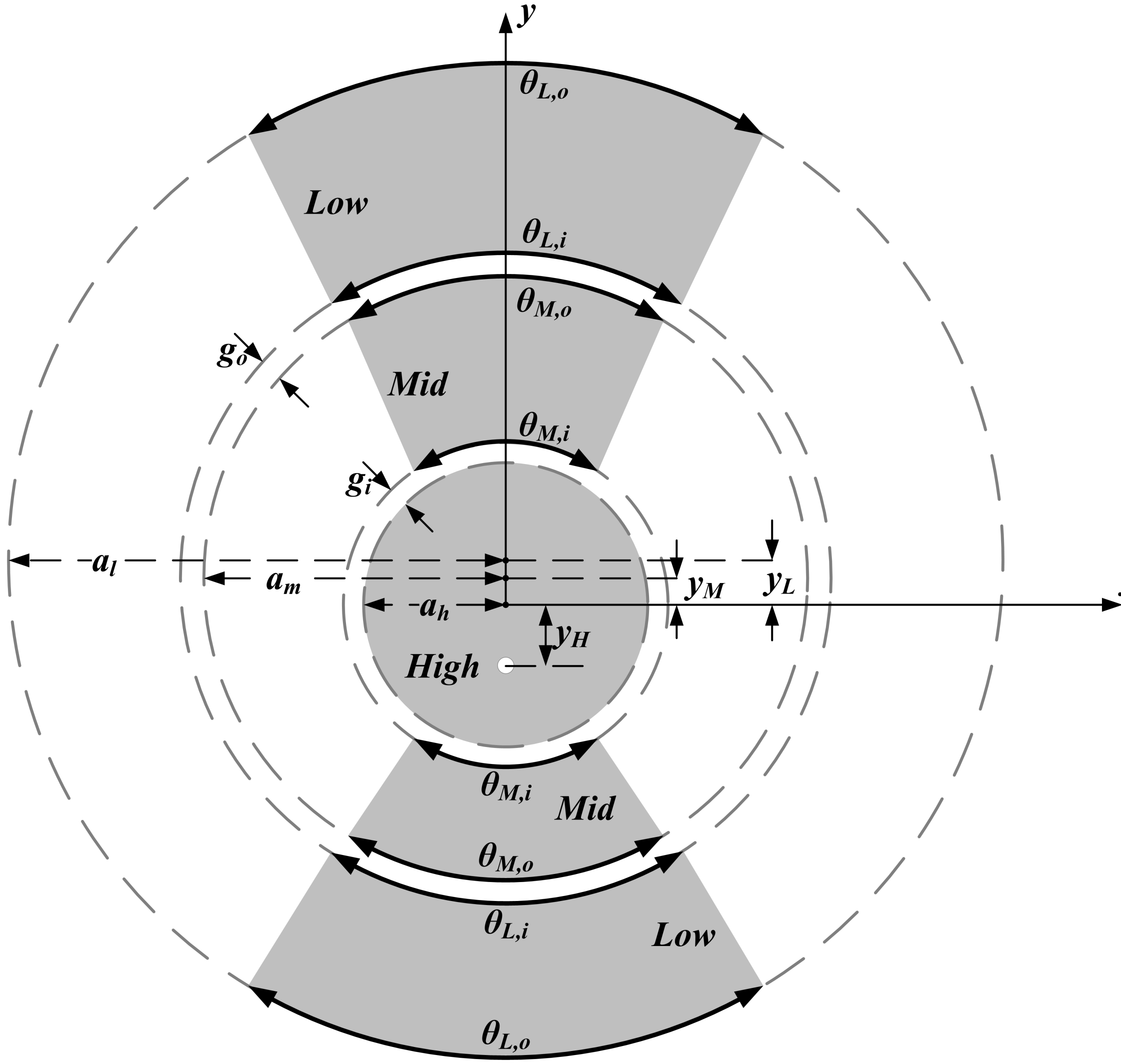


Fig. 3.4: TBPFRA design parameters

sectors, respectively, on the circles defined by radii $a_m + g_o$ and a_l. The angles $\theta_{M,o}$ and $\theta_{L,o}$ set the arc lengths of the outer edges of the mid- and low-band patch sectors, which determines the radiation resistance of the radiating slots and thus the impedance bandwidth of the radiating mode. Wider angles also result in higher coupling between the two orthogonal arms, so these parameters are adjusted during tuning to achieve the widest impedance bandwidth possible while maintaining low coupling to the orthogonal arm.

3.2.2 Reconfiguration Mechanisms

3.2.2.1 Polarization & Band Switching

To actuate the polarization & band switching reconfiguration mechanism, a superstrate is laid over the antenna element containing a set of two fluid channels. The channels are fed from the backplane through the dielectric substrate of the antenna through a set of pumping ports. Fig. 3.5 shows the layout of the fluid channels on the front side of the antenna element. The channels are filled with a continuous-phase dielectric fluid such as Hydrocal 2400 severely hydrotreated napthenic oil, as well as a set of plug inclusions of a liquid metal such as eutectic Gallium Indium alloy (eGaIn). The eGaIn plugs are spaced in the continuous-phase fluid such that they align with the gaps in the arms. The fluid in the channel makes direct contact with the copper sectors of the antenna, This system comprises the polarization & band switching network (PBSN). By applying differential pressure to the fluid channel using a peristaltic pump, the eGaIn plugs can be displaced in the PBSN and reconfigure the polarization state and operating frequency band of the antenna. The control mechanisms for each channel of the PBSN are illustrated in Fig. 3.6. Note that the reservoir & pump segment on the antenna element backplane has significantly more volume than the fluid channel in the superstrate. This allows two separate sets of eGaIn plugs to be

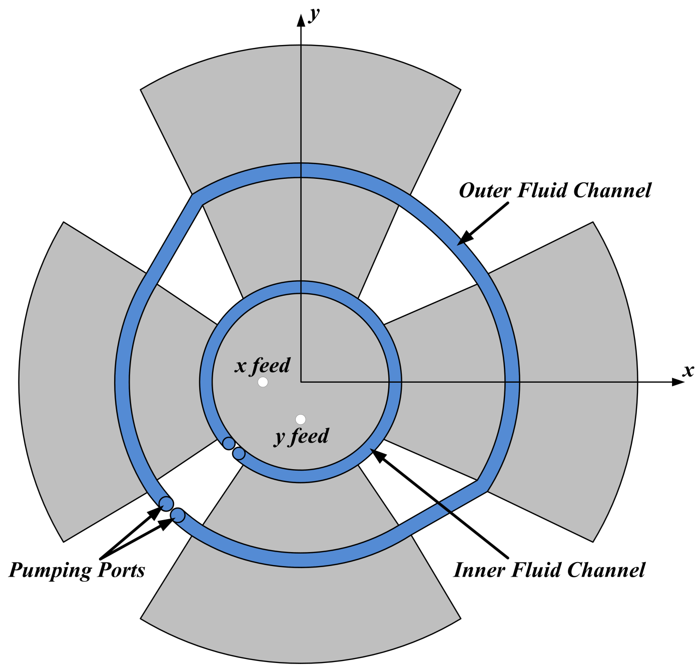


Fig. 3.5: TBPFRA polarization & band switching network fluid channel layout

handled: one set to connect one arm individually, and one set to connect both arms simultaneously. A set of conductive fluid sensor probes in the reservoir segment are used to provide positional feedback from the eGaIn plugs to the controller operating the peristaltic pump. With this PBSN configuration, and a phase switching network (capable of feeding each port individually, or feeding both ports with a 90° phase offset) connected to the probe feeds, the antenna element can be switched through 9 different operating modes, as shown in Fig. 3.7. While additional configurations of the eGaIn plugs are possible, they are not considered in the analysis of this design as they produce identical operating behavior to the modes already depicted in Fig. 3.7.

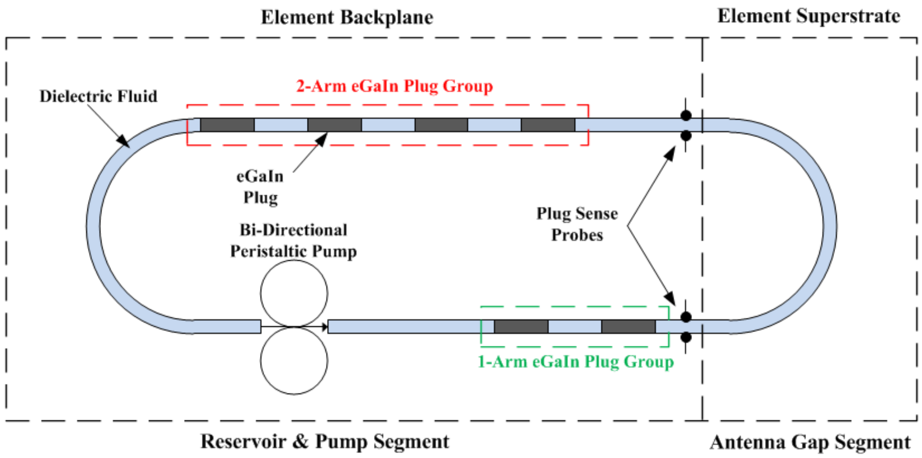


Fig. 3.6: PBSN control & sensing schematic (each fluid channel, 2 total)

3.2.2.2 Impedance Bandwidth Tuning

To achieve tuning of the impedance bandwidth within each operating band, a set of probe-fed COSMIX are connected to the TBPFRA. One COSMIX is attached to each patch sector, at a point equidistant from the inner and outer edge along either the x- or y-axis, allowing frequency tuning within each operating frequency band. The configuration of the COSMIX elements on the TBFPRA is shown in Fig. 3.8 & 3.9. The COSMIX enable operating frequency reconfiguration by applying a variable reactive load to the antenna element. By varying the relative dielectric constant, ϵ_r, of the fluid in the COSMIX, the reactance presented to the antenna element is varied. The dielectric constant of the EFCD in the COSMIX can be varied by varying the ratio of EFCD flow from two reservoirs. By filling one reservoir with a low dielectric constant fluid (i.e. Fluorinert FC-70) and the other with a fluid with high dielectric constant (a Fluorinert/BSTO EFCD), the ratio of pump speeds can control the effective dielectric constant of the mixed flows. This mixed flow can then be directed to a specific COSMIX element by means of a controllable valve network. Such a

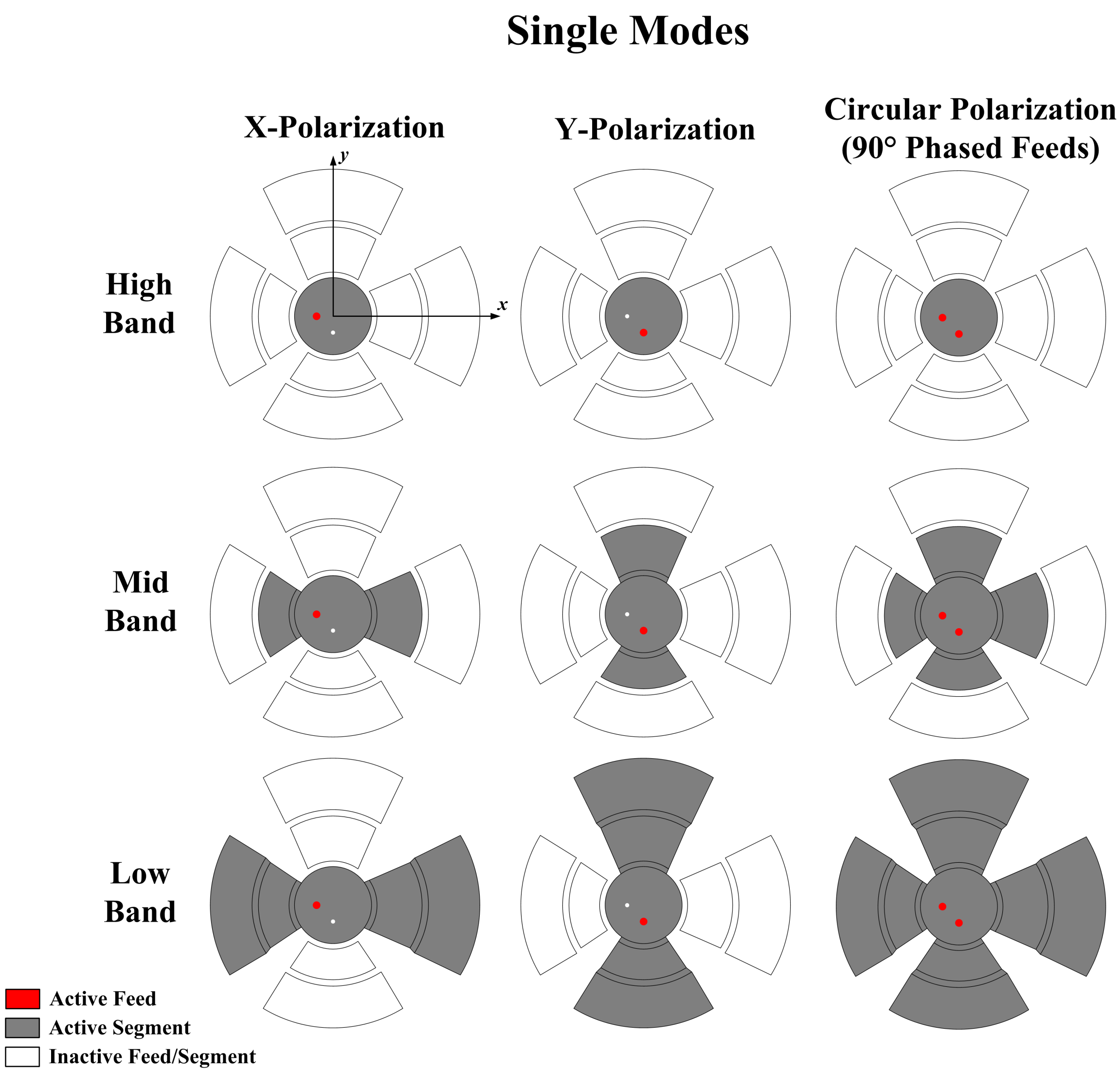


Fig. 3.7: TBPFRA operating modes

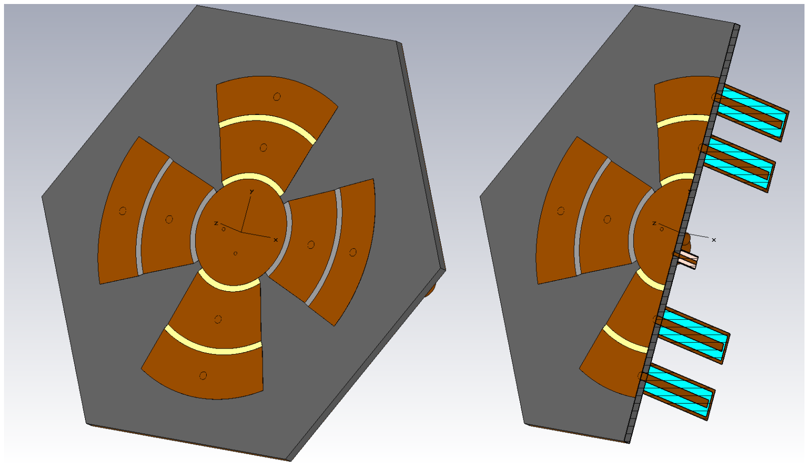


Fig. 3.8: TBPFRA model front with COSMIX

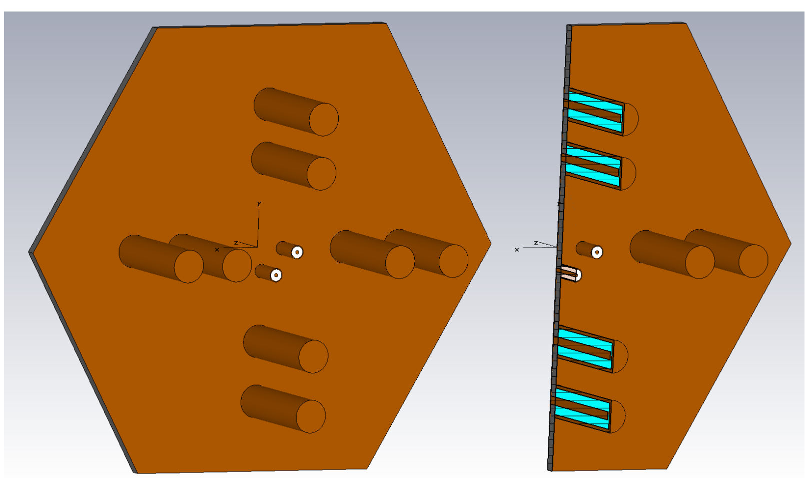


Fig. 3.9: TBPFRA model rear with COSMIX

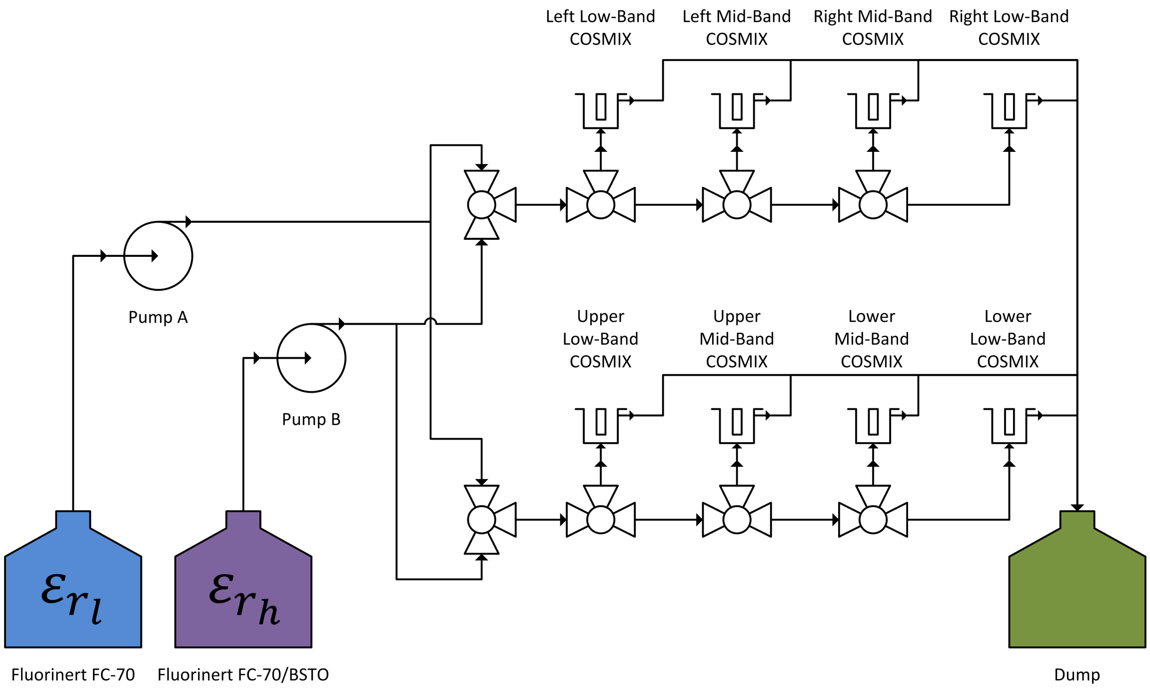


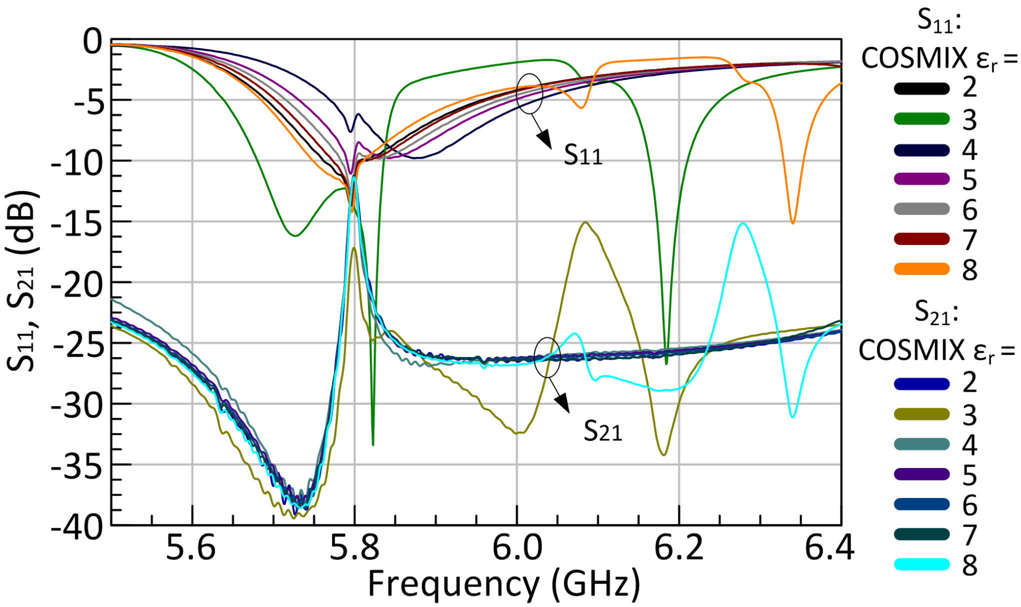
Fig. 3.10: COSMIX fluid control network for TBPFRA

network layout is shown in Fig. 3.10.

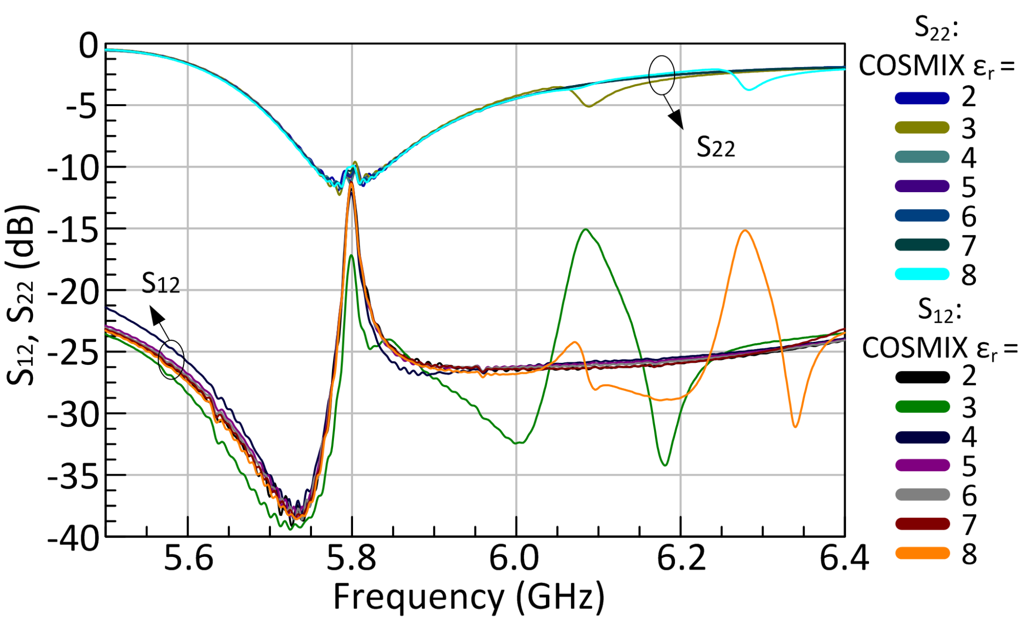
3.3 Simulation

The initial full-wave model of the TBPFRA was built in Ansys Electromagnetics' *HFSS* 3D electromagnetic simulation suite. A companion model was developed in CST's *Microwave Studio* 3D simulation product. A set of simulations were run using the *Microwave Studio* model to evaluate the band switching, polarization reconfiguration, and tuning performance of the TBPFRA. As full-system fluidic reconfiguration testing with the TBPFRA had been discontinued in favor an electronically reconfigurable antenna design, these simulations were run primarily to serve as a rough survey of the operating behavior of the TBPFRA geometry.

To decrease the time necessary to run the several dozen simulations required to survey the TBPFRA in all its operating modes, CST's *Transmission Line Method*



(a) X-feed return loss (S_{11}) and coupling to Y-feed (S_{21})



(b) Y-feed return loss (S_{22}) and coupling to X-feed (S_{12})

Fig. 3.11: TBPFRA high-band simulation: tuning ε_r in inner & outer x-arm COSMIX elements

(TLM) solver was used for the full-wave simulations. The TLM solver works by discretizing the modeled geometry with a hexahedral mesh, and modeling each hexahedral mesh element as a matrix of 12 transmission lines interconnecting the six faces of the hexahedron. The fields in the discretized structure are then modeled as fields propagating through a matrix of the transmission line elements, where the material properties and geometry of each hexahedral cell determines the impedances of its associated transmission lines. Although the TLM solver does not necessarily produce results with the same level of accuracy as, say, the finite-difference time-domain (FDTD) or finite-integration-technique (FIT) full-wave methods, it gives reasonably accurate results with a considerably shorter simulation time. For the simulations presented here, the TLM method took roughly 15 wall-clock minutes to simulate a single excitation on a dual 6-core Xeon E6520 workstation versus roughly 40 minutes for an FIT model with a similarly fine discretization.

Fig. 3.11 shows the simulation survey results from the high-band mode, which was tuned to operate at 5.8GHz. In this set of simulations, ε_r of all four X-arm COSMIXes was varied from $\varepsilon_r = 2$–8. Fig. 3.11a shows the return loss of the X-polarized high-band mode (S_{11}) and coupling from the X-feed into the Y-feed (S_{21}). As can be seen from the S_{11} traces, the mid- and low-band sections of the X-arms do have a loading effect on the high-band mode, and as the X-arm COSMIXes are tuned it results in a change in the X-polarized modal frequency. Conversely, Fig. 3.11b shows the return loss (S_{22}) and coupling to the X-feed (S_{12}) for the Y-polarized mode as excited by the Y-feed. Despite the change in the load on the mid- and low-band X-arms from the COSMIXes, the center frequency of the Y-polarized mode stays the same, with only a change in the coupling to the X-feed as the loading from the COSMIXes changes.

Fig. 3.12 shows the simulation results for the X-polarized mid-band mode. The mid-band geometry was roughly tuned to operate at 3.5GHz with COSMIX $\varepsilon_r = 2$.

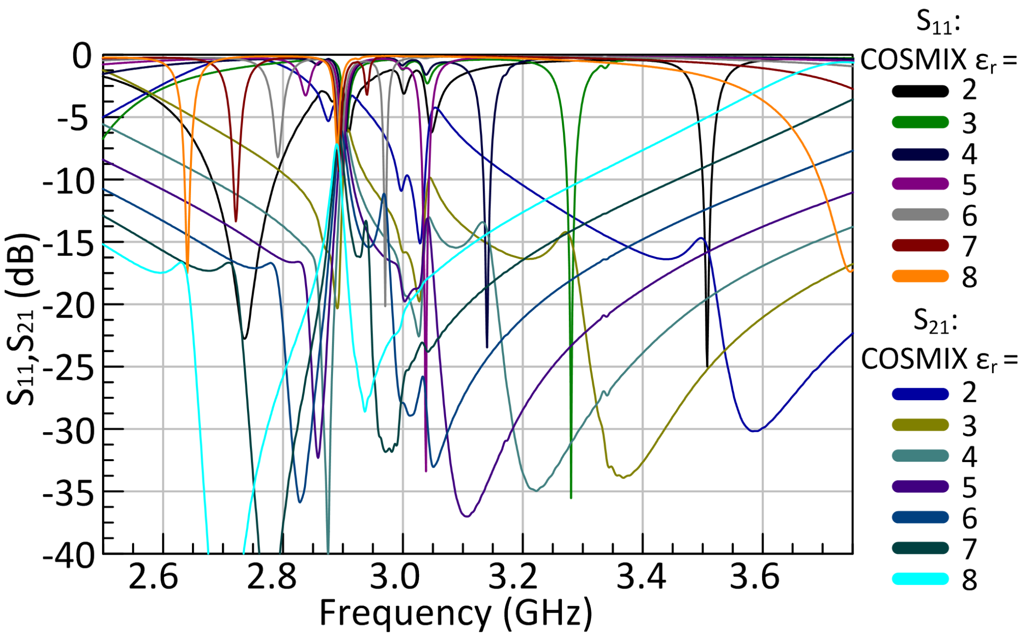


Fig. 3.12: TBPFRA mid-band simulation: tuning ε_r in inner x-arm COSMIX elements

Here, only the inner X-arm COSMIXes had their ε_r varied to explore the tuning behavior of the mid-band mode. Once again, S_{11} is the return loss from the X-feed, and S_{21} is the coupling from the X-feed to the Y-feed. Here, the inner COSMIXes are directly coupled into the active antenna structure, and the tuning of the operating mode is readily apparent. A ε_r range of 2–8 gives a tuning range of roughly 3.5GHz–2.65GHz, or a tuning range of roughly 24% relative to the high end. Of note also is the undesired mode at roughly 2.9GHz. This mode appears to be a non- or poorly-radiating mode, which results in significantly higher coupling from the X-feed to the Y-feed.

Fig. 3.13 shows the simulation results for the X-polarized low-band mode. The low-band mode was roughly tuned to operate at 1.6GHz with $\varepsilon_r = 2$. As in Fig. 3.11, here ε_r for the inner & outer X-arm COSMIXes was varied from $\varepsilon_r = 2$–8. Once

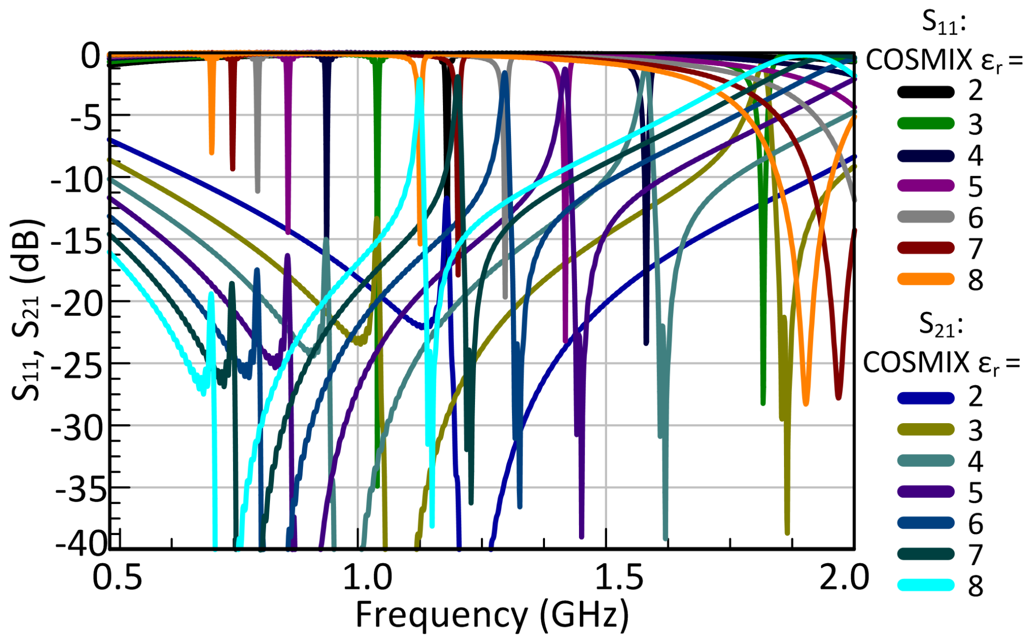


Fig. 3.13: TBPFRA low-band simulation: tuning ε_r in inner & outer x-arm COSMIX elements

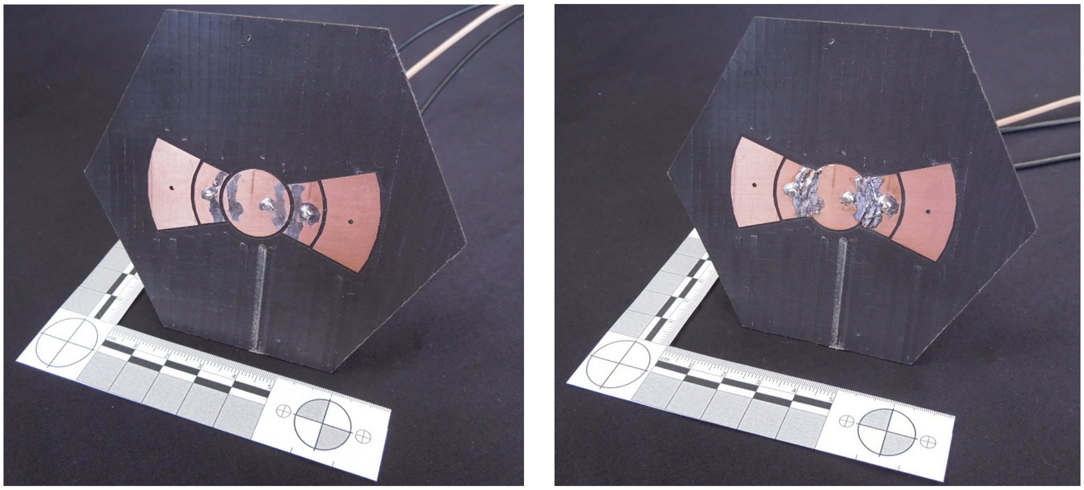
again, because the COSMIXes are directly coupled to the low-band structure, significant tuning of the operating mode is observed. Here, a tuning range of roughly 1.25GHz–0.65GHz is observed, a tuning range of 48% relative to the high end. Of note here is that a higher-order mode starts to appear at higher ε_r values, manifesting as a second dip in S_{11} and a corresponding sharp rise and subsequent dip in S_{21}. In particular, above $\varepsilon_r = 7$ this higher-order mode tunes into the range of the fundamental low-band mode. The corresponding peak in the X-feed to Y-feed coupling at this higher-order mode suggests that, like the undesired mode in the mid-band results, it is not an effective radiating mode and induces high coupling between the feed ports.

The results of this simulation survey are meant to show the wide range of tunability and reconfigurability offered by the TBPFRA. Clearly, there is significant room to improve this antenna design—in particular to reduce coupling between the orthogonal probe feeds, and to optimize the impedance tuning of the operating modes across their individual frequency tuning ranges. As the goal of this work was to demonstrate system-level integration and control of these advanced fluidic tuning mechanisms in a multifunctional antenna array, the effort required to fully optimize the design of the TBPFRA was instead devoted to the development of the overarching system. These simulations are presented to demonstrate the potential of the TBPFRA.

3.4 Fabrication & Testing

3.4.1 Electromagnetic Tests

A prototype of a single-arm TBFPRA was built to demonstrate the band switching and frequency tuning mechanisms. In this prototype, copper tape soldered over the gaps was used to emulate the effect of the liquid metal, and Hittite Microwave HMC928LP5E analog phase shifters [49] were used to emulate the effect of the COSMIX to achieve frequency tuning. Fig. 3.14a shows the fabricated prototype configured for high-band operation. Fig. 3.14b shows the copper tape & solder bridges over the inner gaps, used to emulate the liquid metal for prototype testing. In this configuration, the antenna operates in the mid-band mode. The alternating black & white blocks on the ruler in Fig. 3.14 are each 1cm long. Fig. 3.15 shows the HMC928LP5E evaluation boards attached to SMA connector probes used to emulate the reactive loading of COSMIX on the antenna element. The phase shifters accept a DC bias voltage of 0–12V and generate a phase delay of 450° which reduces to roughly 0° at 12V bias. Thus, at 12V applied bias the open-circuit-terminated phase shifter board appears electrically similar to a COSMIX with a low ε_r dielectric. When the bias voltage is reduced, the electrical length of the phase shifter increases,



(a) High-band configuration (b) Mid-band configuration

Fig. 3.14: Fabricated single-arm TBFPRA prototype

a behavior analogous to that of a COSMIX as its ε_r is increased.

Without phase shifters attached to the SMA probes on the outer sectors, the prototype TBPFRA exhibits the input return loss characteristics shown in Fig. 3.16a. The fabricated design was tuned to operate at 1 GHz, 3 GHz, and 6 GHz without external loading. In Fig. 3.16a, the high-band mode shows a clear match at roughly 6.1 GHz, the mid-band mode shows a good match at roughly 2.95 GHz, and the low-band mode is matched at 1.5 GHz.

Fig. 3.16b shows the measured return loss with the phase shifters attached and biased to 0V. In comparison to Fig. 3.16a, loading of high-band mode by the phase shifters on the mid-band arm segments can be seen. This behavior matches that observed in the high-band simulation shown in Fig. 3.11 as the COSMIX ε_r was increased.

With the antenna configured as in Fig. 3.14b, the bias voltage applied to the phase shifter was varied. The results of this tuning test are shown in Fig. 3.17. Note first that, since the antenna was designed without any reactive loading mechanism,

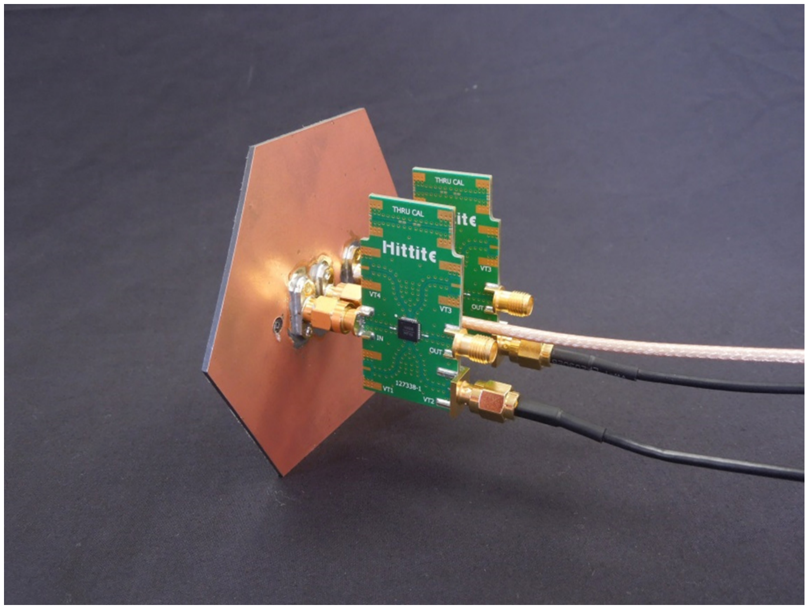
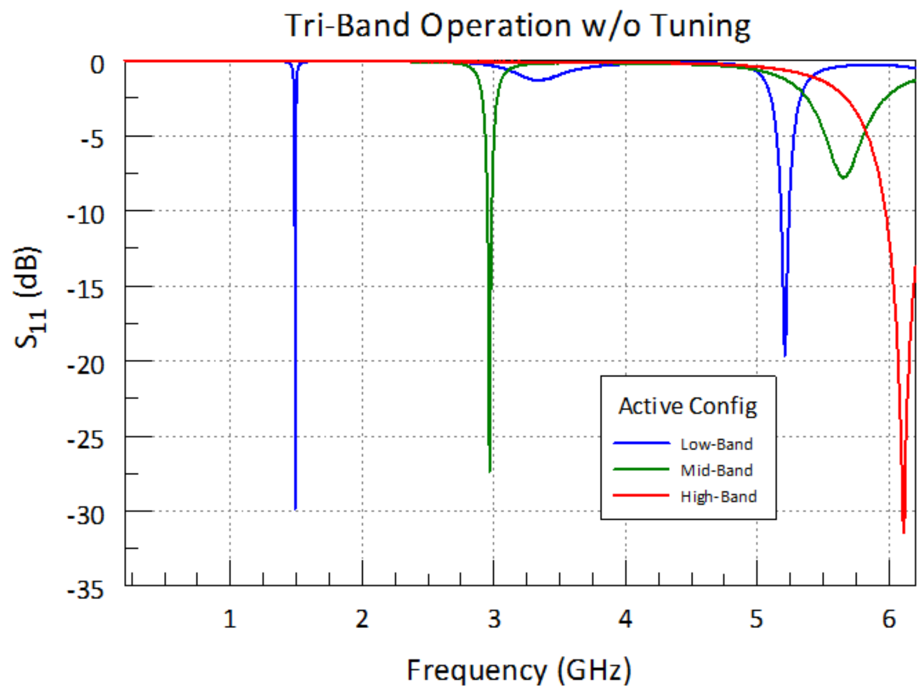


Fig. 3.15: Analog phase shifters used to emulate COSMIX

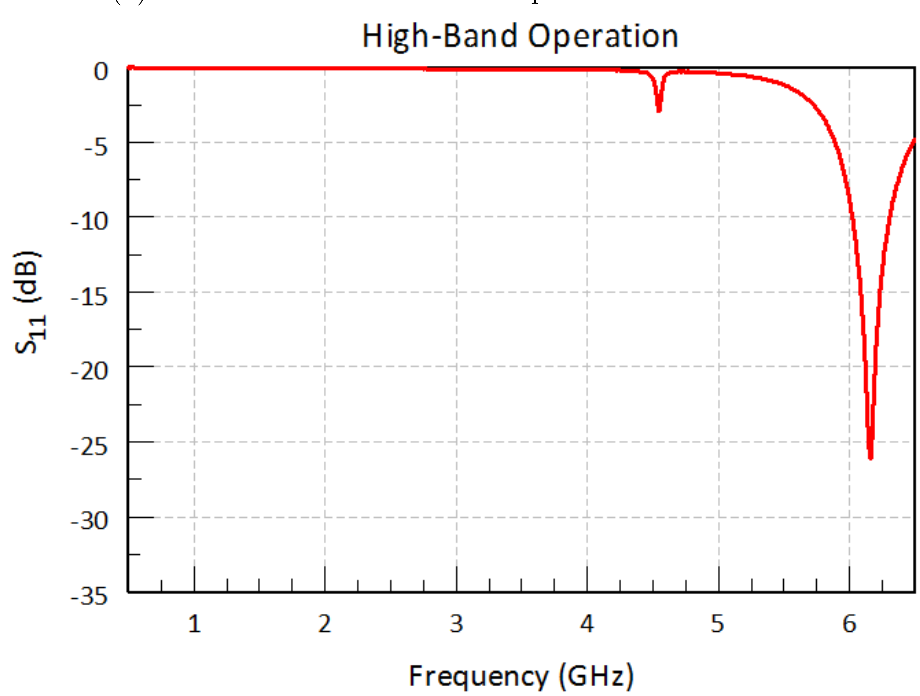
the presence of the phase shifter's loading effects dropped the mid-band operating frequency to roughly 1.7GHz at 12V bias (minimum phase shift). Despite this shift in operating frequency, which is to be expected, variable frequency tuning is observed as the phase shifter's bias voltage is varied.

3.4.2 Thermal Tests

To test the performance of the thermal reconfiguration system, a prototype heat exchanger was designed to fit the backplane of the TBPFRA prototype. For the first tests, the probe feeds were omitted for simplicity. The prototype heat exchanger is shown in Fig. 3.18. This heat exchanger was adhered to the back of a fabricated single-arm TBPFRA element using *Sylgard 184* PDMS silicone encapsulant, and connected to a thermal control loop filled with circulating Fluorinert FC-70. The thermal system was commanded to slew to temperatures above and below ambient, and the resulting element temperatures were profiled with a FLIR Systems T440



(a) Measured TBPFRA without phase shifters attached



(b) Measured TBPFRA with phase shifters, high-band mode

Fig. 3.16: TBPFRA prototype operation without & with phase shifters

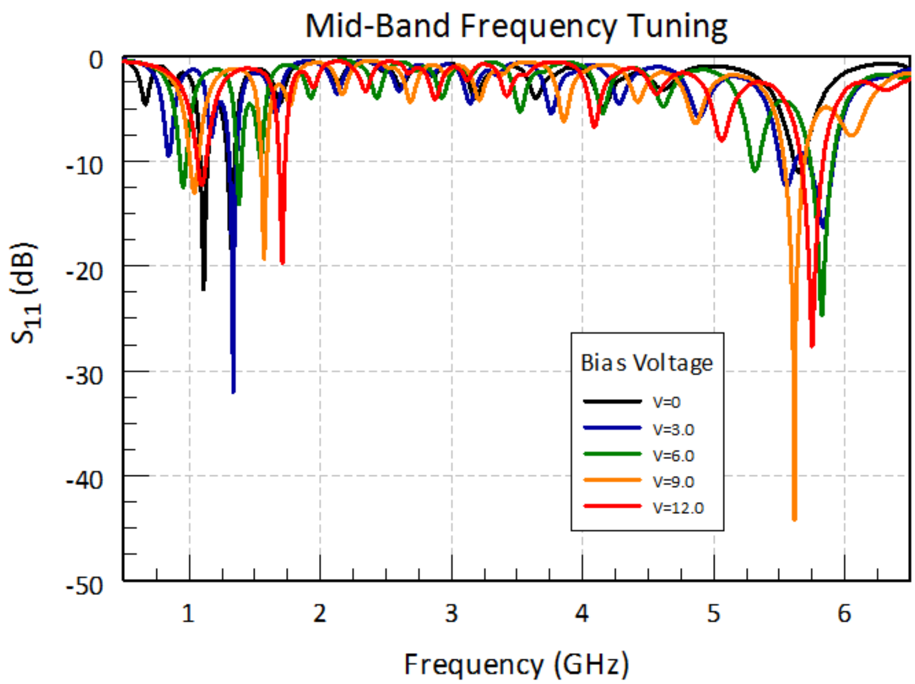


Fig. 3.17: Measured TBPFRA with phase shifters, tuning mid-band mode

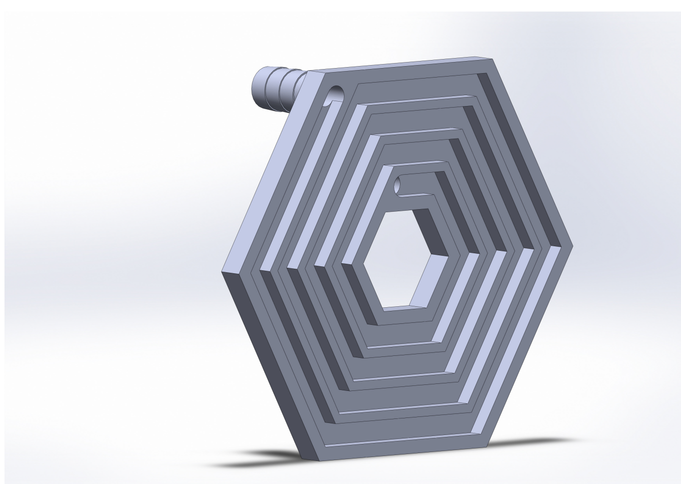
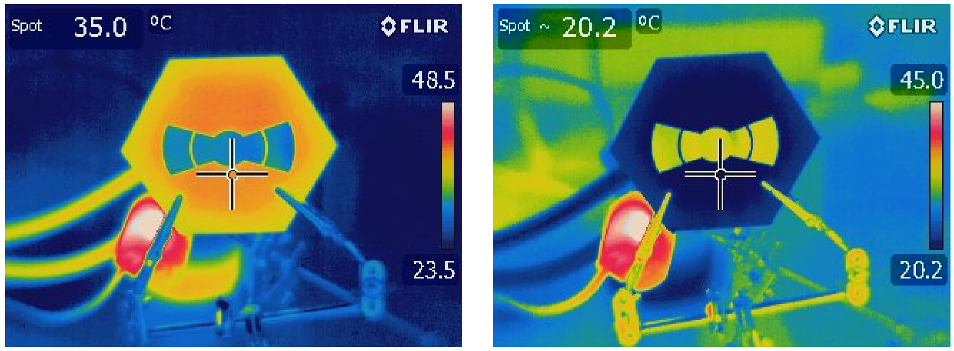


Fig. 3.18: TBPFRA heat exchanger design



(a) Heating to 35°C (b) Cooling to 20°C

Fig. 3.19: Thermal control system testing at 25°C ambient temperature

thermal imaging camera. The results of this test are shown in Fig. 3.19.

3.4.3 Liquid Metal Tests

Several strategies were explored to implement the fluid network to route the PBSN over the front of the antenna element. Several dielectric fluids and channel materials were tested for their compatibility with eGaIn and the feasibility of manipulating isolated plugs of eGaIn as in Fig. 3.6. Hydrocal 2400, silicone (PDMS) oil (500 centistoke and 1000 centistoke viscosities), and Fluorinert FC-70 fluorocarbon oil were tested as continuous-phase motive fluids for an eGaIn plug in a pressure-driven fluid channel consisting of 1.58mm ID PTFE (Teflon) tubing. Two distinct phenomena were observed during these tests. The lower viscosity fluids (Fluorinert FC-70: 12 centistokes) failed to push the eGaIn as a plug but instead flowed past the eGaIn. The higher viscosity oils (Hydrocal & PDMS) successfully moved the eGaIn as a plug, but left a residue as shown in Fig. 3.20. Fig. 3.20 shows optical microscope images of the debris left after a 0.2mL eGaIn plug was pushed back and forth through a 1.58mm ID PTFE tube roughly 15 times.

This debris has not been fully characterized, but its continued generation in

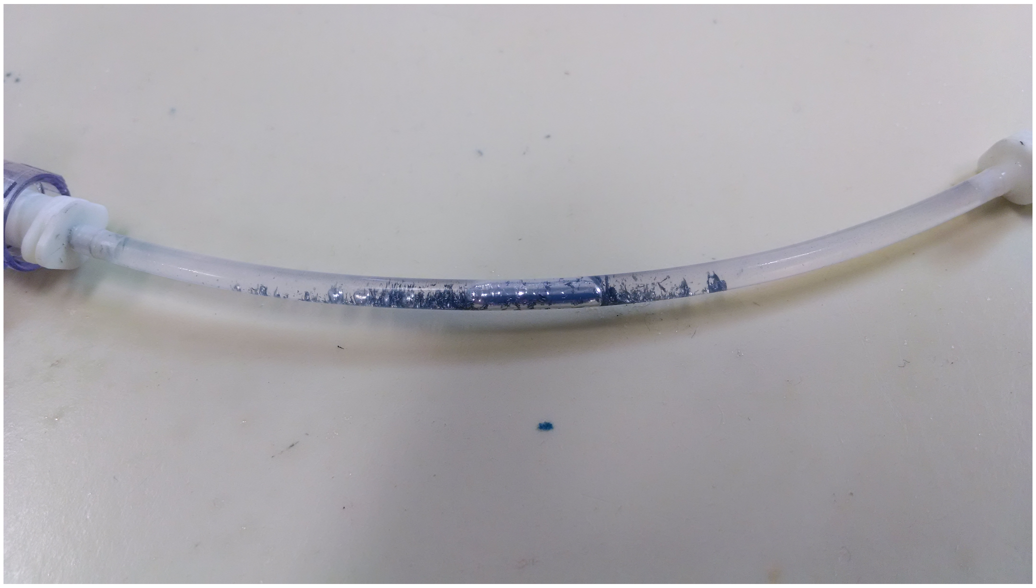
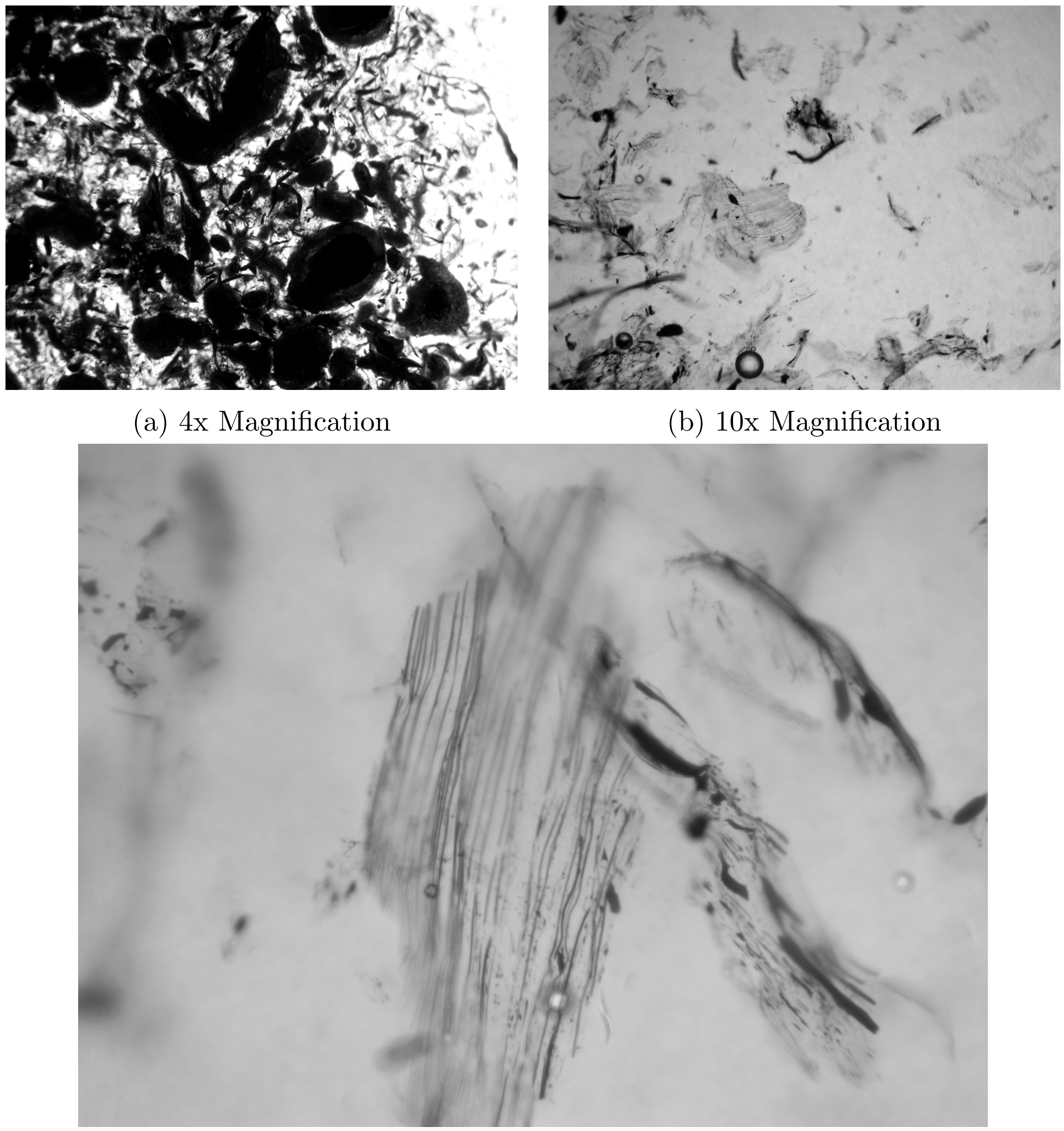


Fig. 3.20: eGaIn plug in 1000 cst silicone oil showing debris sloughing

dozens of trials suggests that different material handling strategies need to be developed to utilize eGaIn in a pressure-driven network in this manner. It is known that eGaIn forms a "skin" of gallium oxide–Ga_2O_3–on exposure to ambient oxygen, which deforms plastically unlike the liquid eGaIn below. [50] It is also known that silicone oil has a relatively high oxygen solubility. [51] Thus, it is hypothesized that during plug flow conditions the gallium oxide skin around the eGaIn plug is stressed until it separates and exposes unoxidized eGaIn to the continuous-phase silicone fluid, which likely contains a significant amount of dissolved oxygen. On exposure to the silicone fluid, the fresh eGaIn rapidly forms more gallium oxide, increasing the total volume of the oxide skin. Once enough oxide has formed that flakes begin to protrude from the surface of the eGaIn, forces due to interfacial flow at the eGaIn/silicone boundary tear the oxide flakes away from the eGaIn, resulting in the debris seen in Fig. 3.20.

The observation of this behavior in the liquid metal fluid system led to the pursuit of an alternate reconfiguration technique for use in the full array system



(a) 4x Magnification (b) 10x Magnification

(c) 40x Magnification

Fig. 3.21: Optical microscopy of eGaIn debris in silicone oil

4. ELECTRONICALLY POLARIZATION-RECONFIGURABLE ANTENNA (EPRA)

4.1 Design Goals

The goal for the electronically polarization-reconfigurable antenna (EPRA) design is to achieve a compact, planar, switchable-polarization antenna element which can be integrated into a hexagonal array and controlled by the multifunctional antenna reconfiguration control system. Unlike the TBPFRA discussed in section 3, the EPRA design is not intended to leverage a frequency reconfiguration mechanism, but is instead designed to operate at a single frequency in the unlicensed 2.4GHz industrial, scientific, and medical (ISM) frequency band. Furthermore, as the name implies, the EPRA is designed to leverage an electronic reconfiguration mechanism—PIN diodes—to achieve polarization switching between two orthogonal polarization states. An electronic reconfiguration approach was chosen due to the maturity of PIN diode technology compared to the fluidic mechanisms explored in section 3. A planar form factor was chosen to facilitate the EPRA's integration with the thermoregulation system as well as the planar hexagonal array topology.

4.2 Design

What follows is a discussion of the design & implementation of the EPRA element. First, an overview of the concept inspiring the geometry is presented. Following this, an overview of the PIN diode-based polarization reconfiguration mechanism is presented.

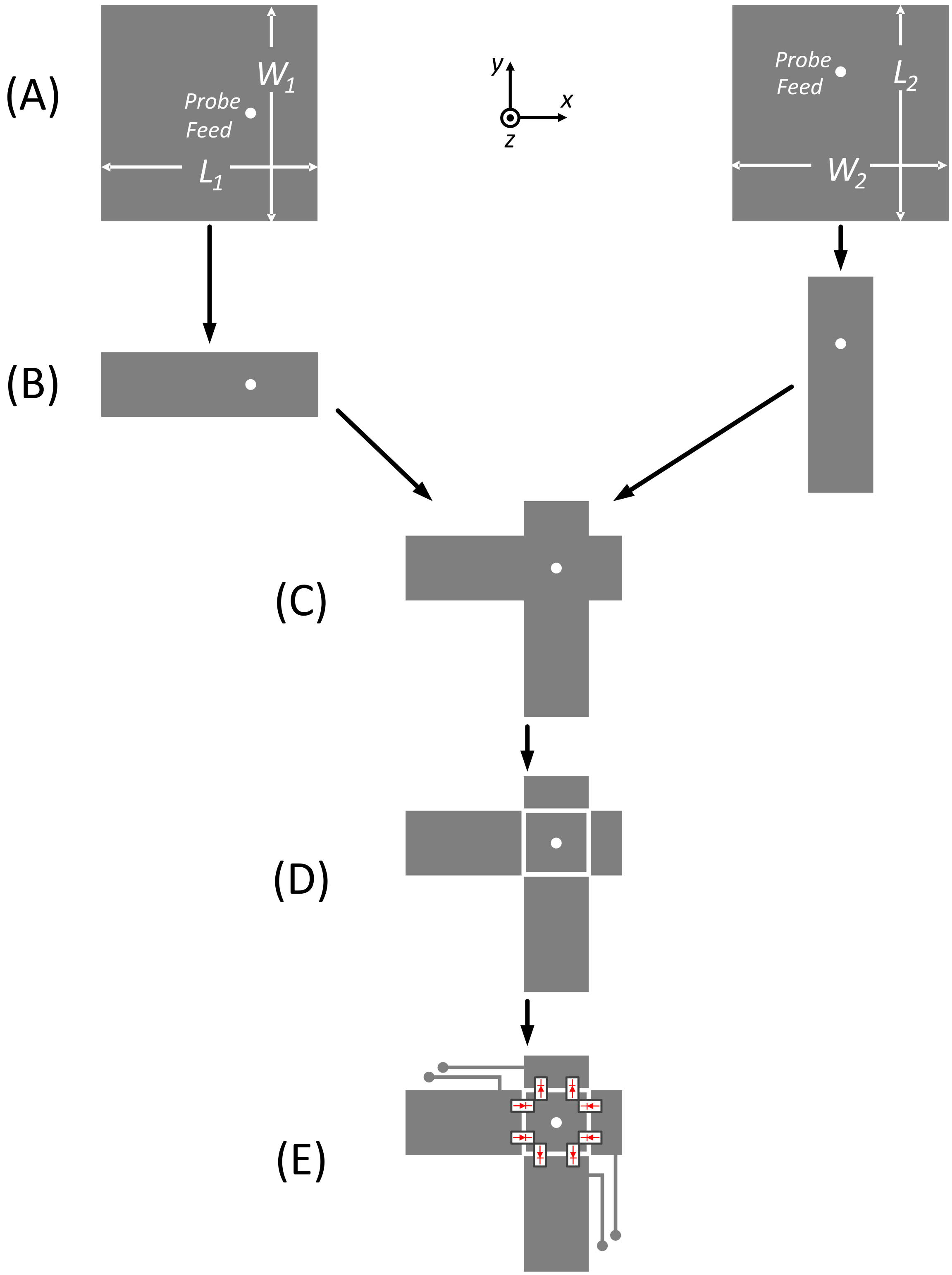


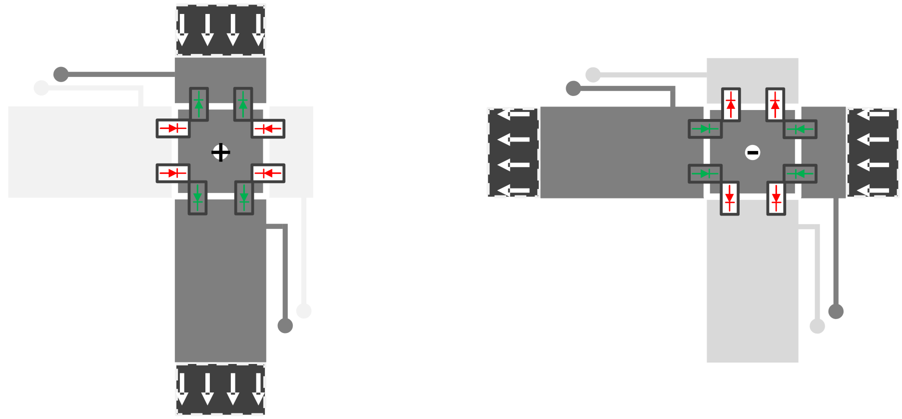
Fig. 4.1: EPRA design concept

4.2.1 Concept

Fig. 4.1 shows a graphical overview of the concept underlying the EPRA design. The geometry of the EPRA element is inspired by a set of orthogonal rectangular microstrip patches. As discussed in section 2.1.1.3, a rectangular microstrip patch is fundamentally a linearly-polarized antenna structure. The radiated electric field of a rectangular patch is linearly polarized parallel to the resonant length L of the patch geometry. In part (A) of Fig. 4.1, two such rectangular patches are shown, where the left patch is resonant and polarized along the x-axis and the right patch is polarized along the y-axis. In part (B), the orthogonal patches have their non-resonant width dimensions (W_1 and W_2) reduced. This width reduction adversely affects the radiation efficiency of the antennas as it significantly reduces the width of the patches' radiating slots, but the operating frequency stays roughly the same. The two orthogonal, narrow patches are overlaid such that they share a common probe feed in part (C). In this state, both patches are in parallel and excited simultaneously by the feed. Thus, in part (C) a gap is cut around the probe feed to separate the two sets of "arms." Finally, in part (E) a set of narrow, high-impedance DC bias lines are used to connect the isolated arms to DC ground (with vias to the ground plane), and a set of RF PIN diodes are attached across the gap. The diodes are connected at the outer edges of each arm because an analysis of the currents on the rectangular patch shows that the RF current is highest parallel to the resonant length L along the outer, non-radidating edges of the patch.

4.2.2 Reconfiguration Mechanism

In this EPRA design, the set of RF PIN diodes (Skyworks SMP1345 in SC-79 surface-mount packages) bridging the gaps between the arms are used as a set of current-controlled resistors to reconfigure the polarization of the antenna element.



(a) Positive bias voltage: y-polarization (b) Negative bias voltage: x-polarization

Fig. 4.2: DC bias controls EPRA polarization state

When each PIN diode is forward-biased with $I_f = 10$mA, it presents a series RF resistance of $R_f \approx 1.5\Omega$. In a reverse-biased state, the junction capacitance $C_r \approx 0.16$pF of the diode is presented across the gap, effectively isolating the outer patch arms from the central feed. Fig. 4.2 shows this reconfiguration process graphically. In Fig. 4.2a, a positive DC bias voltage is applied between the center conductor and the ground plane. This forward-biases the four PIN diodes on the vertical arm and activates the Y-polarized configuration. When a negative DC bias voltage is applied, as in Fig. 4.2b, the diodes on the horizontal arm are forward-biased and the X-polarized configuration is active.

The bias lines shown in Fig. 4.1 & 4.2 are designed to be very narrow width, so that they act as very high impedance microstrip lines. Further, The length of the line between the EPRA arms and the via through the substrate to the ground plane is chosen such that the line is roughly $\lambda_g/4$ at the operating frequency of the EPRA. Thus, the bias line appears, from the perspective of the EPRA element, to be a short-circuit terminated transmission line of length $\lambda_g/4$. From transmission-line theory, we know that a short-circuited transmission line presents an input impedance [52]

$$Z_{\text{in}} = jZ_0 \tan \beta l \tag{4.1}$$

When the line is a quarter-wavelength long such that $l = \lambda_g/4$, we have $\beta l = \pi/2$ and thus $Z_{\text{in}} \to \infty$ in (4.1). Thus, the bias lines appear as an open circuit at the antenna element, and they minimally perturb the fields in the EPRA's patches while still providing a return path for the DC bias current for the PIN diodes.

To inject the DC bias onto the coaxial transmission line feeding the EPRA element, a commercially-available DC bias tee (Pasternack Enterprises PE1615) is used. Since the reconfiguration control system only uses a single-sided DC supply, a ground-isolation scheme is employed to provide the reversible-polarity bias voltage necessary to properly bias both polarization states of the EPRA. In this scheme, the ground plane and outer conductor of every EPRA element in the array is isolated from the other elements. This ground isolation is accomplished by the use of an insulating support structure for the array elements and a set of outer-conductor DC blocks (Pasternack Enterprises PE8211) on the RF ports of the bias tees. Fig. 4.3 shows the schematic of the implemented biasing network (for each EPRA element in the array). A set of bias & overcurrent protection resistors sets the bias current for the PIN diodes and provides overcurrent protection to the control board in case of an accidental short between two EPRA elements' ground planes. The resistor values are identical, and are calculated according to

$$R_b = \frac{1}{2}\frac{V_{\text{supply}} - V_f}{I_{f,\text{total}}} \tag{4.2}$$

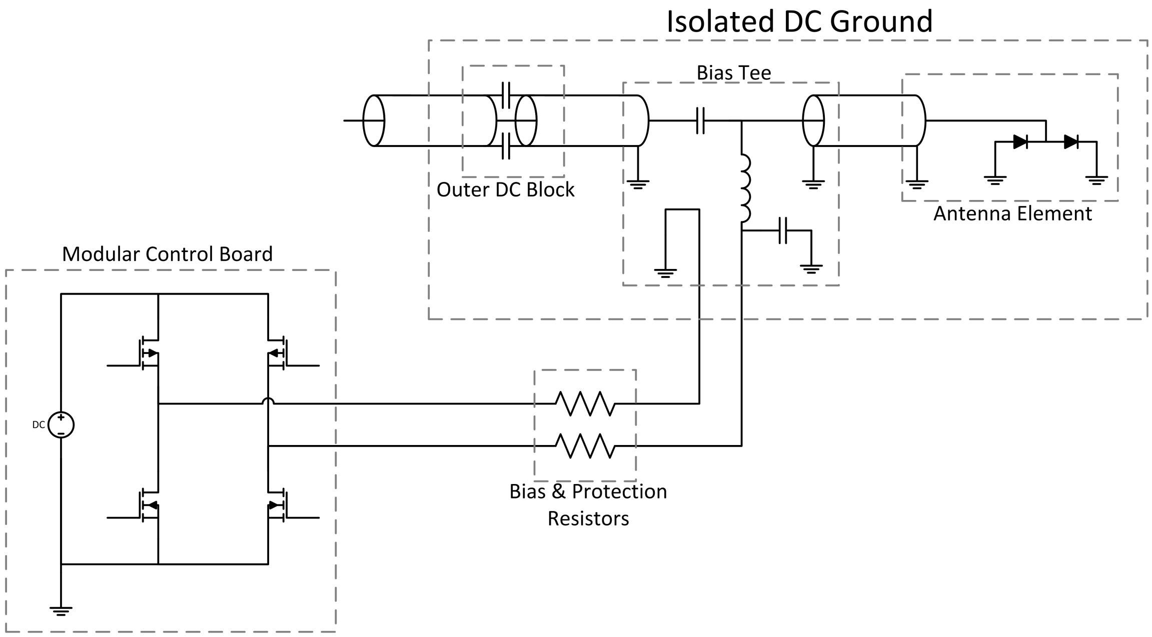


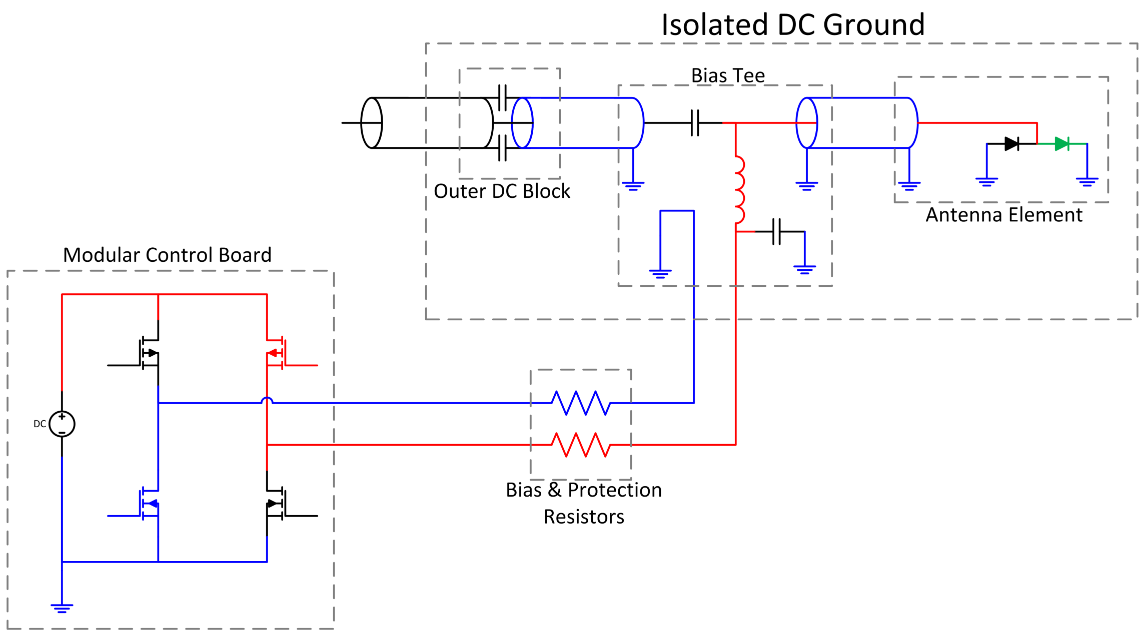
Fig. 4.3: EPRA DC biasing network

where V_{supply} is the DC supply voltage, V_f is the forward voltage drop across the PIN diodes, and $I_{f,\text{total}}$ is the total forward bias current for the set of four diodes on each arm. Overcurrent limiting is accomplished by selecting $V_{\text{supply}} >> V_f$, so that if a short occurs between two oppositely-biased isolated grounds, the fault current will be

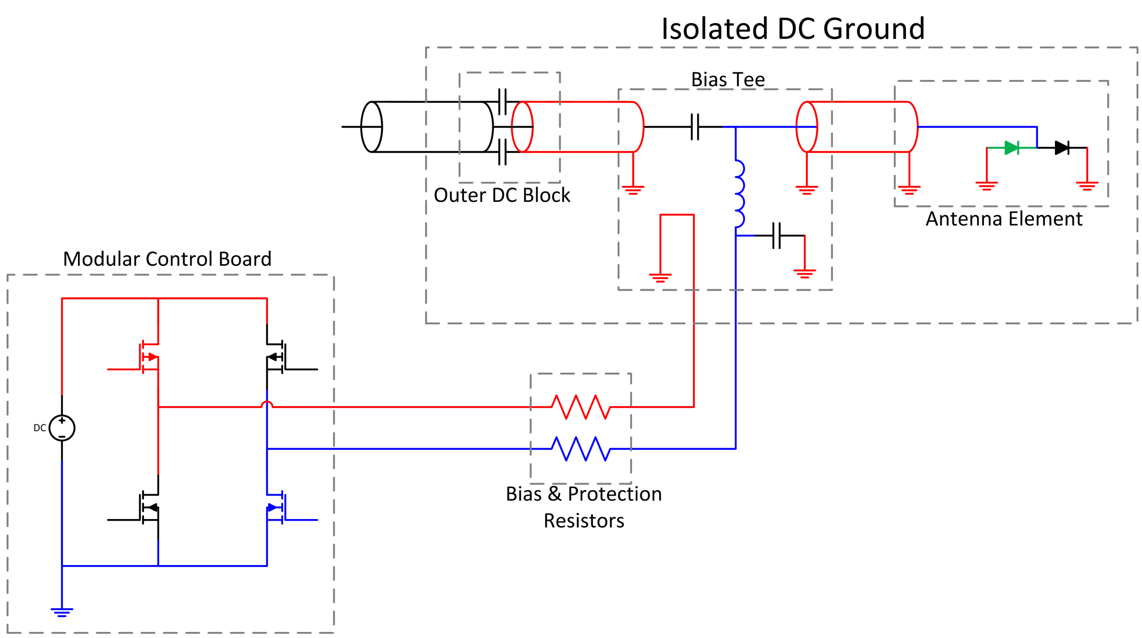
$$I_{\text{fault}} = \frac{V_{\text{supply}}}{2R_b} \approx I_{f,\text{total}} \tag{4.3}$$

Thus, the resistors will limit the current to roughly the same value ($\approx$ 40mA) in both normal biasing conditions and in worst-case fault-to-ground conditions.

Fig. 4.4 shows a schematic of the bias network in operation. An H-bridge on the modular controller board supplies a reversible-polarity voltage of roughly 16VDC. In the Y-polarization state, +16V is supplied to the center conductor of the EPRA's feed line and the outer conductor is grounded to 0V. In the X-polarization state, the



(a) y-polarization state



(b) x-polarization state

Fig. 4.4: DC bias operation (red: positive, blue: negative)

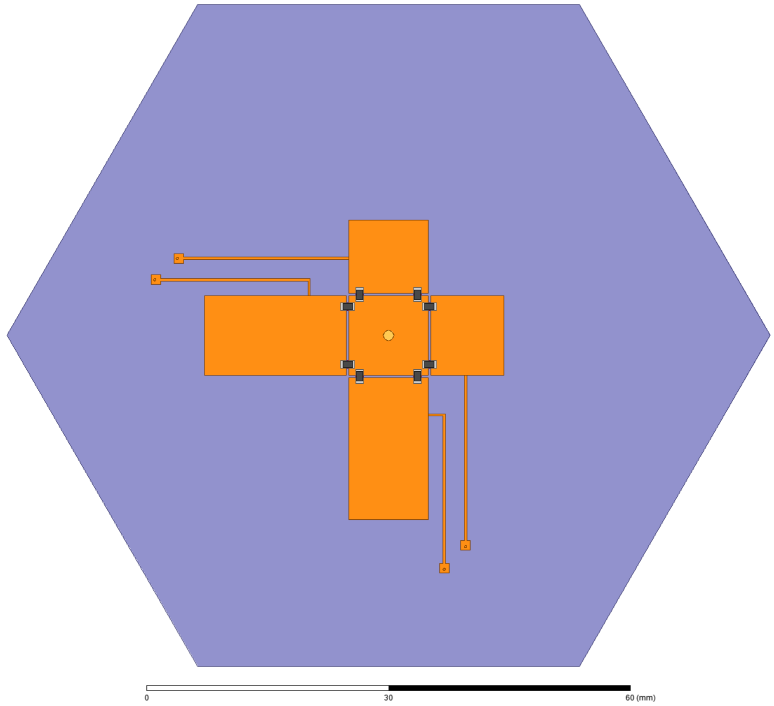
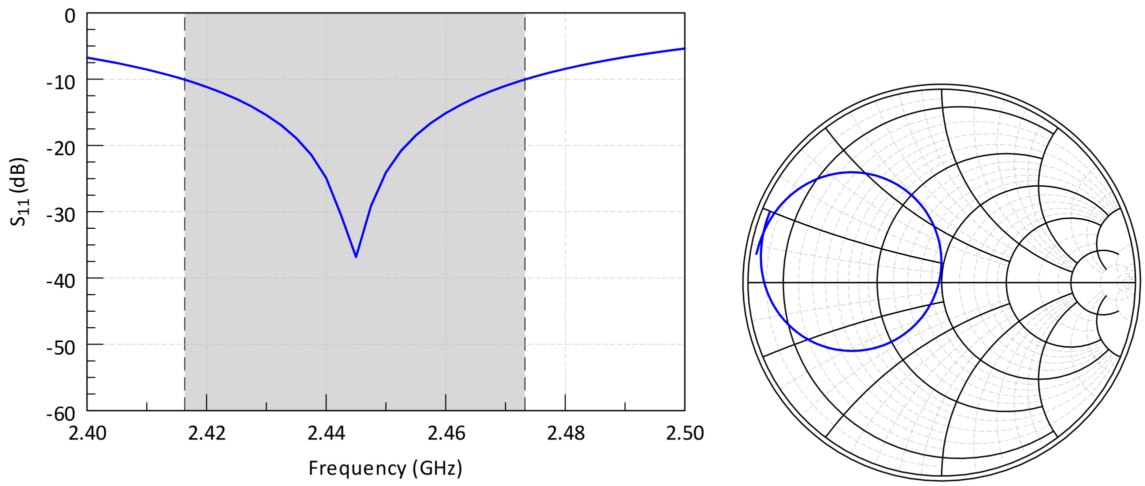


Fig. 4.5: EPRA element *HFSS* model

center conductor is grounded to 0V and the isolated outer conductor is biased to +16V. The value of the bias resistors is chosen as $R_b = 200\Omega$.

4.2.3 Modeling & Simulation

To verify the operation of the EPRA design, a model was assembled in Ansys Electromagnetics' *HFSS* 3D EM simulation software. The final model is shown in Fig. 4.5. The PIN diodes are modeled as a combination of solder & plastic blocks to represent the SC-79 packages and impedance boundaries to model the PIN junction. The solder & plastic are used to capture the effect of package & mounting parasitics, and the impedance boundaries are used to model the parasitic inductance of the junction & bond-wires as well as the forward resistance R_f/junction capacitance $C_{j,r}$ depending on whether the diode is modeled in the forward- or reverse-biased state.

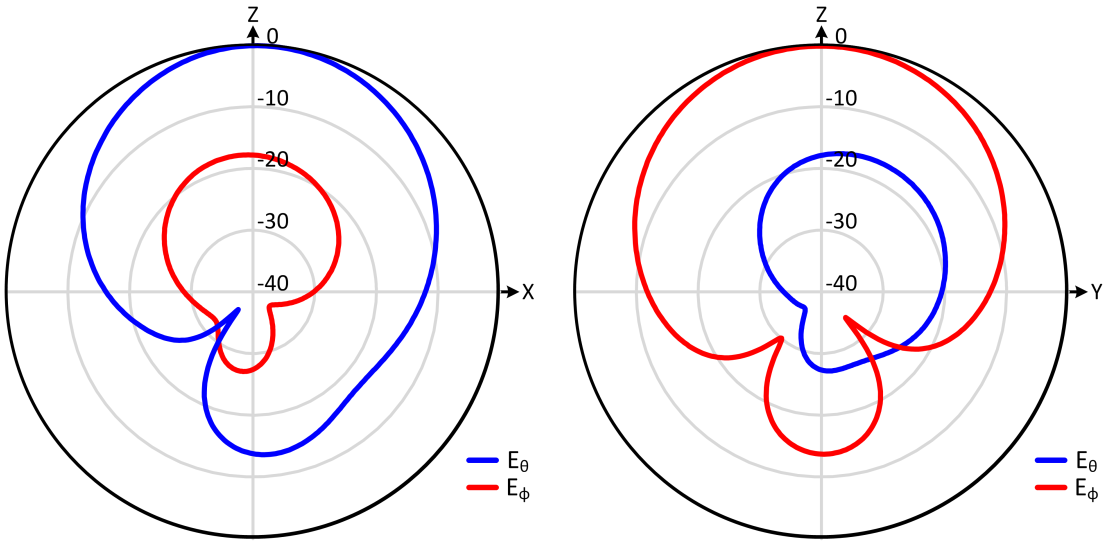


(a) Return loss

(b) Input impedance

Fig. 4.6: Simulated EPRA impedance behavior: x-polarized mode

The model was simulated and tuned to operate at roughly 2.45GHz, in the center of the 2.4GHz ISM band. A set of simulations was run to analyze the EPRA element's impedance & radiation behavior. Fig. 4.6 shows the impedance behavior of the simluated model. The dip in the magnitude of the reflection coefficient shown in Fig. 4.6a indicates that the antenna is tuned to operate at 2.445GHz. This is similarly indicated by the input impedance as plotted on the impedance Smith chart as shown in Fig. 4.6b, where the input impedance at the operating frequency is $Z_{in} = 48.6 + j0.04\ \Omega$ (where 50Ω is located at the exact center of the chart). The observed simulated 2:1 VSWR impedance bandwidth is 50MHz, which is 2.0% of the 2.445GHz center frequency. The simulated radiation patterns of the X-polarized mode are shown in Fig. 4.7. The element shows reasonable polarization purity with a roughly 18dB cross-polarization ratio in both the $\phi = 0°$ and $\phi = 90°$ planes.



(a) $\phi = 0°$ plane (b) $\phi = 90°$ plane

Fig. 4.7: Simulated EPRA radiation patterns: x-polarized mode

4.3 Fabrication & Testing

A set of seven EPRA elements were fabricated on hexagonal tiles of Rogers RT/Duroid 5880 substrate using a T-Tech, Inc. QuickCircuit 5000 milling machine. The EPRA elements were mounted in a planar hexagonal array configuration in a sheet of 10mm thickness ROHACELL HF polymethylacrylimide structural foam. ROHACELL HF is a low dielectric constant ($\varepsilon_r = 1.05$), low-loss ($\tan\delta \approx 2 \times 10^{-4}$) rigid foam with excellent RF, insulating, and mechanical rigidity characteristics. The ROHACELL foam facilitates the DC isolation of the EPRA ground planes as discussed in section 4.2.2. Fig. 4.8 shows a close view of the center EPRA element as mounted in the array structure.

4.3.1 Electromagnetic Tests

The fabricated EPRA array was first tested to ensure that the individual EPRA elements were performing as expected. A set of return loss measurements were made

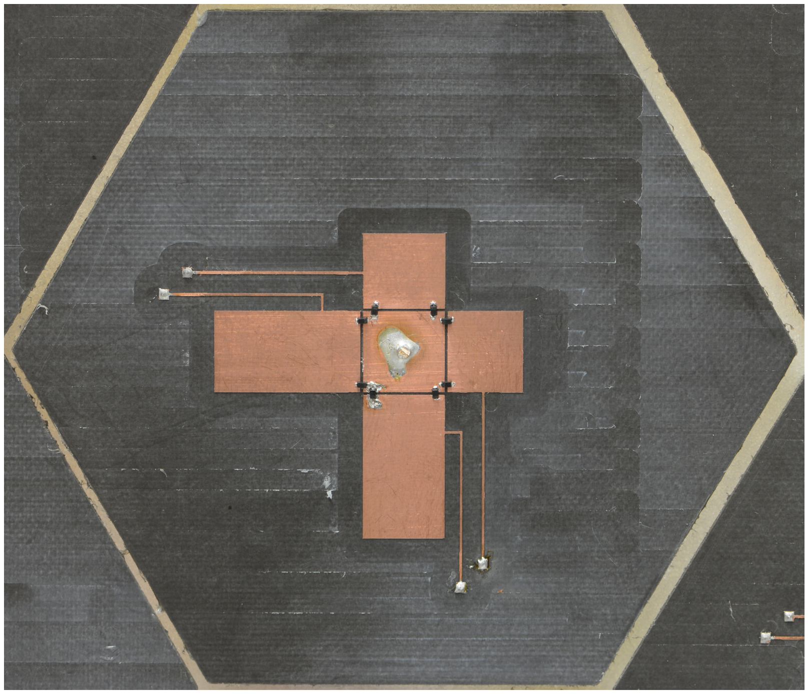
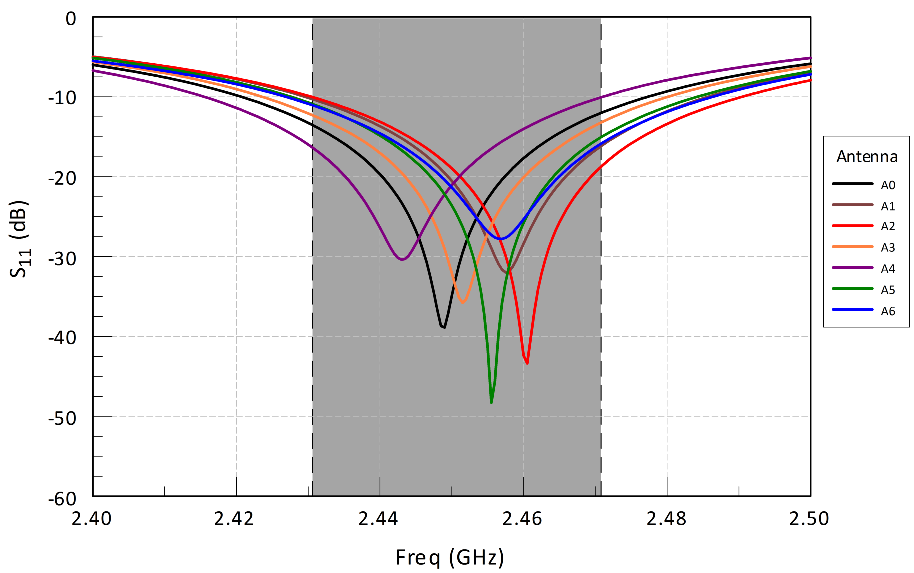


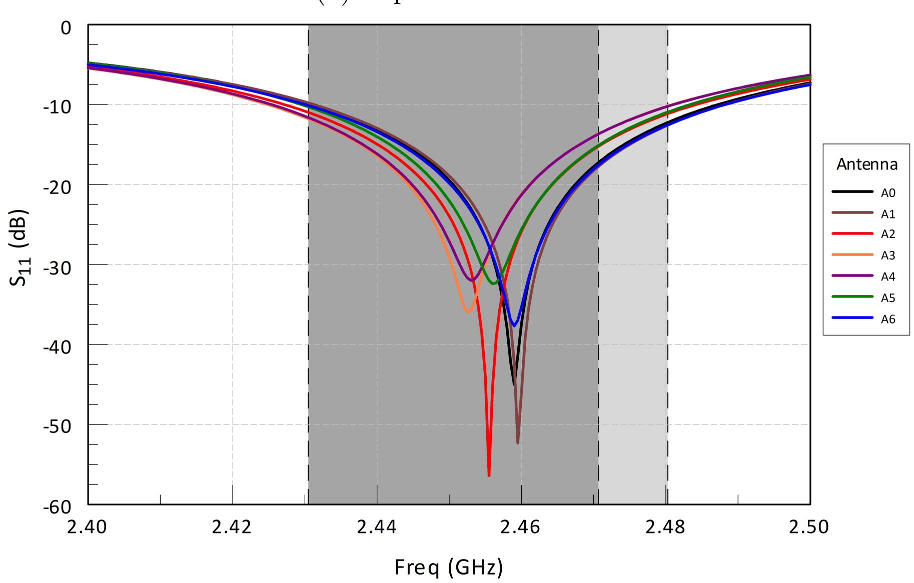
Fig. 4.8: Fabricated EPRA element mounted in hexagonal array

with an Agilent Technologies E8361C programmable network analyzer in both polarization states to confirm the elements' operation. Fig. 4.9 & 4.10 show these measurements. In Fig. 4.9, the effect of manufacturing variance can be seen on the operating frequencies of the individual antenna elements. The X-polarized modes in Fig. 4.9a show slightly more variation than the Y-polarized modes in Fig. 4.9b. The region highlighted in dark grey shows the 2:1 VSWR bandwidth common to both polarization states, and the light grey region in Fig. 4.9b shows the additional impedance bandwidth of the Y-polarized mode due to the tighter grouping of that mode's operating frequencies. For the set of seven, the common impedance bandwidth is roughly 40MHz, or 1.6% of the 2.46GHz center frequency.

Next, the radiation pattern of the center element in the array was measured in an anechoic chamber to verify the radiation behavior of the simulated model. Fig. 4.11 &

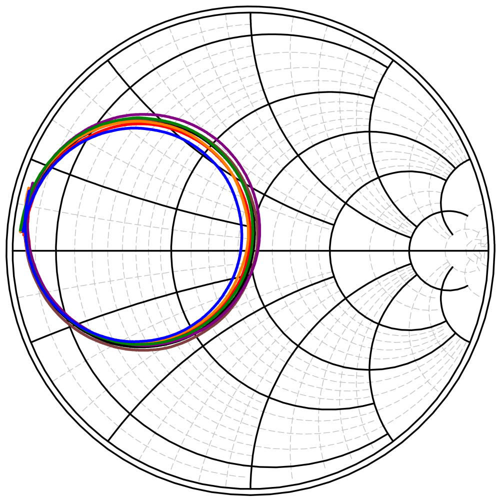


(a) X-polarized modes

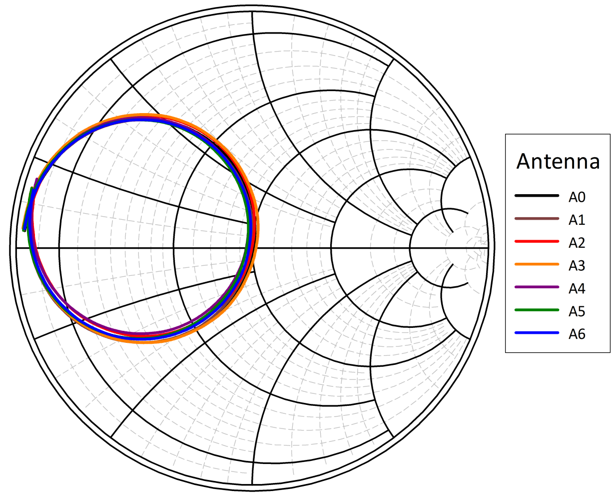


(b) Y-polarized modes

Fig. 4.9: Return loss of fabricated EPRA elements



(a) x-polarized modes



(b) y-polarized modes

Fig. 4.10: Input impedance of fabricated EPRA elements (measured from 2GHz–3GHz, normalized to 50Ω)

4.12 show the measured radiation patterns of the X-polarized & Y-polarized modes, respectively. Comparing Fig. 4.11 to Fig. 4.7, it can be seen that the fabricated antenna is performing quite close to the modeled behavior. The measured cross-polarization ratio is slightly degraded from that of the simulation, although this is could be attributable to scattering off the adjacent antenna elements (which were terminated in matched loads during the pattern measurement) as well as the control electronics & support structure.

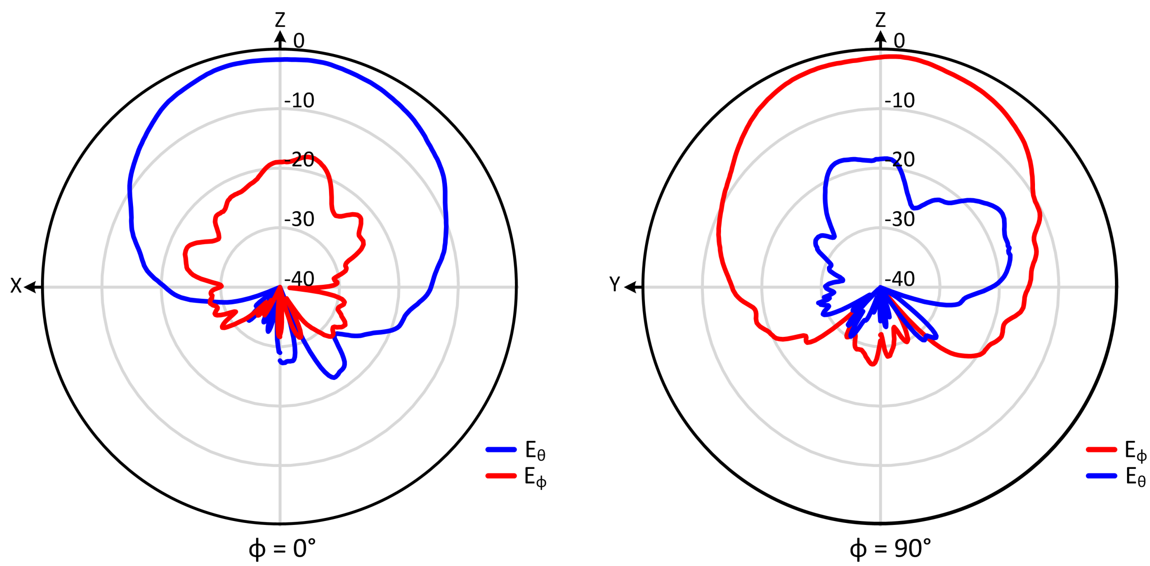


Fig. 4.11: Measured EPRA element radiation pattern: x-polarized mode

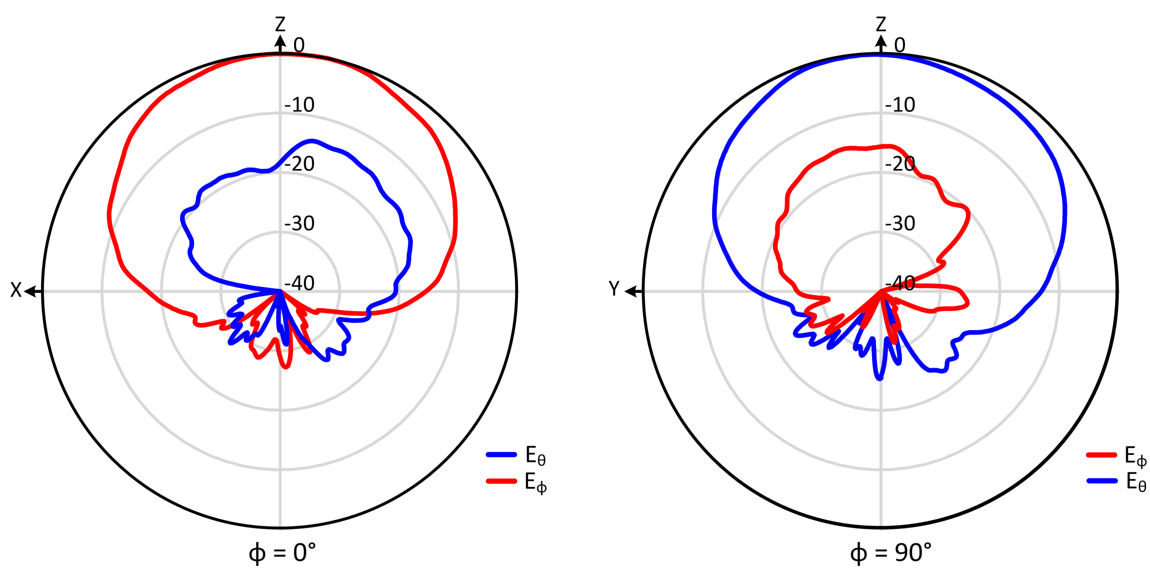


Fig. 4.12: Measured EPRA element radiation pattern: y-polarized mode

5. MODULAR CONTROL BOARD & THERMOREGULATION SYSTEM

5.1 Design Goals

To control both the fluid reconfiguration mechanisms and thermal state of the tri-band polarization & frequency reconfigurable antenna (TBPFRA) design described in section 3, a modular control board (MCB) was designed. The purpose of the MCB is to accept commands sent over a wireless data link from the array control server and respond to those commands by either manipulating the physical state of the TBPFRA/EPRA element or returning measured or stored data about its current state. The MCB has two sets of inputs: a set of conductive fluid sensor inputs that enable positional sensing of the fluids in the polarization & band-switching channels, and a set of thermistor inputs that provide temperature feedback from various points in the thermal control subsystem. The MCB also has two sets of outputs: a set of combined power/PWM control outputs to control a set of servo-actuated valves to direct fluids in the reconfiguration mechanisms of the TBPFRA, and a set of bi-directional pulse-width modulated DC outputs to control the heat flux through a thermoelectric cooler (TEC) and the speed and direction of a set of pumps. The TEC is used to control the temperature of the antenna element, and the pumps supply the motive force to the reconfiguration fluid networks in the TBPFRA. A microcontroller (MCU) on the MCB interfaces with the sensor inputs and control output peripherals. A wireless radio module enables the MCU to interface with a WiFi network to receive commands and send responses. A block diagram of the reconfiguration mechanisms controlled by each MCB is shown in Fig. 5.1.

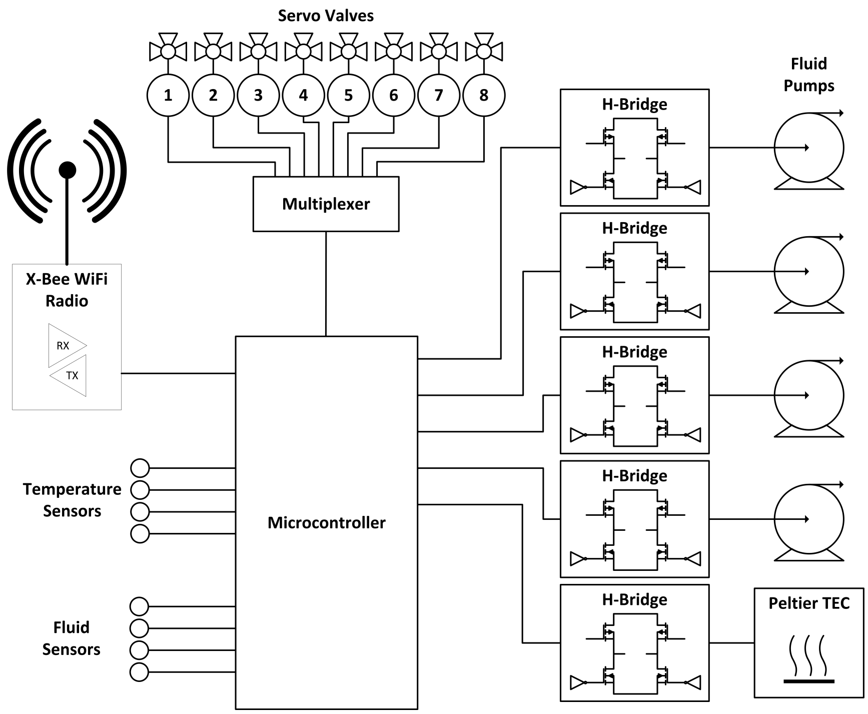


Fig. 5.1: Block diagram of modular control board

5.2 Design

5.2.1 Sensor Inputs

As mentioned in section 5.1, the MCB is furnished with two sets of sensor inputs. A set of four conductive fluid sensors is used to detect the position of the liquid metal plugs in the polarization & band switching network (PBSN). Another set of four thermistor temperature sensors is used to measure the temperature of key components in the thermal system.

5.2.1.1 Conductive Fluid Sensors

The conductive fluid sensors detect the presence or absence of a fluid by measuring the conductivity between two sensor electrodes immersed in the fluid tubing. The physical layout of a fluid sensor is shown in Fig. 5.2. The sensor electrodes consist

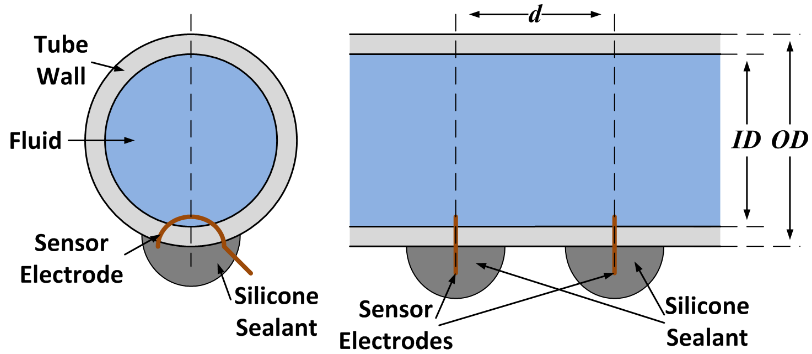


Fig. 5.2: Lateral (left) and axial (right) cross-sections of a conductive fluid sensor in a fluid tube

of AWG 30 (0.254mm diameter) copper wire inserted through the wall of 1/16" ID, 1/8" OD vinyl tubing, separated by a distance d of 1 cm. To ensure fluid-tightness, a drop of either PDMS silicone elastomer (Sylgard 184) or RTV silicone sealant is applied to the outside of the vinyl tube where the copper wire penetrates the tube wall.

The principle of operation of the fluid sensor is as follows. When a liquid metal plug is *not* present, the fluid sensor circuit is as shown in Fig. 5.3. The majority of the circuit (delineated by the dashed rectangle) is located on the MCB, with a pair of wires connecting the sense electrodes in the fluid tubing to the reference voltage source and signal conditioning network. The resistance $R_{e,h}$ is roughly proportional to the bulk resistivity of the dielectric fluid ρ_d multiplied by the electrode separation d, or $R_{e,h} \propto \rho_d\, d$. Since the dielectric fluids of interest have bulk resitivities on the order of $10^{15}\ \Omega \cdot \mathrm{cm}$ [35] , we have $R_{e,h} \approx 10^{15}\ \Omega$. As a plug of liquid metal is pumped past and comes into contact with both electrodes, as shown in Fig. 5.4, the resistance between the sensor electrodes changes to $R_{e,l} \propto \rho_m\, d$, where ρ_m is the bulk resistivity of the liquid metal. Liquid metals of interest such as eGaIn or Hg have resistivities on the order of $10^{-7}\ \Omega \cdot \mathrm{cm}$, which gives $R_{e,l} \approx 10^{-7}\ \Omega$. Thus, the passage of a plug

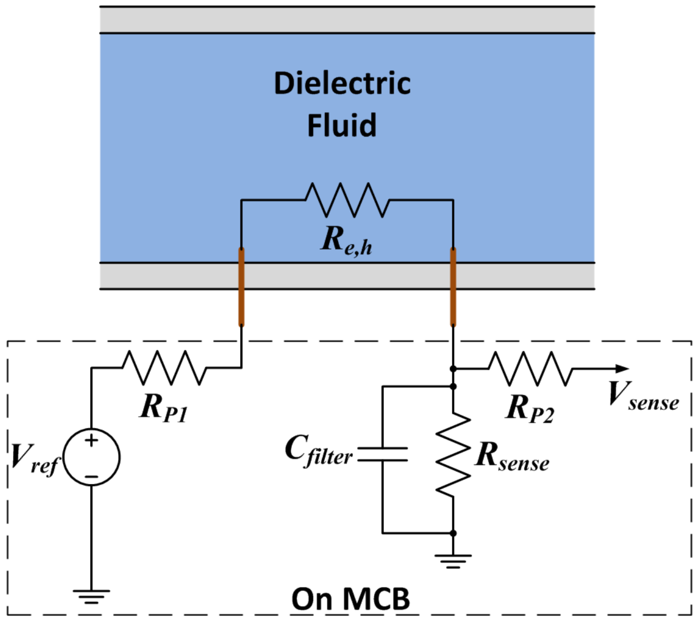


Fig. 5.3: Fluid sensor circuit with only dielectric fluid present

of liquid metal causes a change in resistance between the sensor electrodes of nearly 10^{22} Ω.

V_{ref} in Fig. 5.3 & 5.4 is a regulated 5V supply rail, and V_{sense} is connected to a digital input on the MCU as a liquid metal presence indicator. The digital input has an input impedance $Z_{\text{in}} \approx 5\ M\Omega$, and the threshold voltage, V_{th}, between a logical low (0) and logical high (1) is roughly 1.5V with a supply voltage of 5V [53]. The resistor network consisting of R_{P1}, R_{P2}, R_{sense}, and R_e forms a voltage divider between the equivalent resistances $(R_{P1} + R_e)$ and $(R_{\text{sense}} \| (R_{P2} + Z_{\text{in}}))$, where

$$V_{\text{sense}} = V_{\text{ref}} \frac{R_{\text{sense}} \| (R_{P2} + Z_{\text{in}})}{(R_{\text{sense}} \| (R_{P2} + Z_{\text{in}})) + (R_{P1} + R_e)}. \tag{5.1}$$

To ensure reliable detection of the liquid metal and ensure immunity to any induced noise voltage, it is desirable to have $V_{\text{sense}} \ll V_{\text{th}}$ when no liquid metal is present and $V_{\text{sense}} \gg V_{\text{th}}$ when liquid metal is present, so the values of R_{sense}, R_{P1}, and R_{P2} are

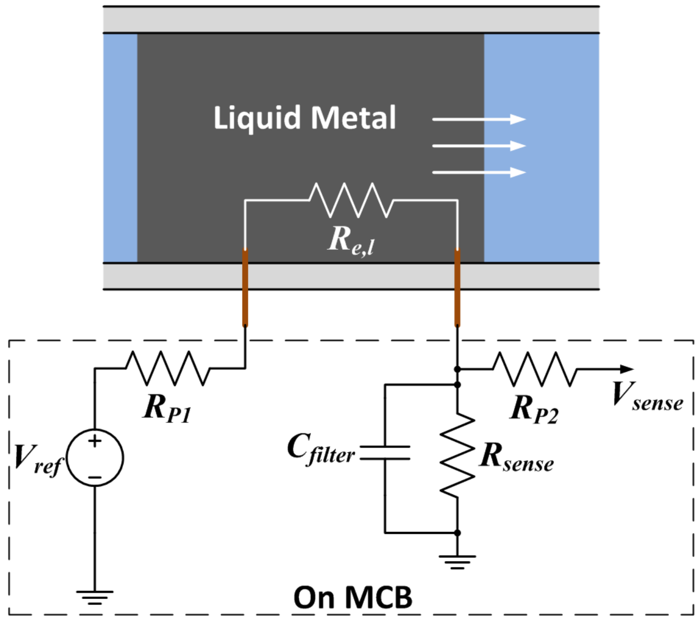


Fig. 5.4: Fluid sensor circuit with passing liquid metal plug

chosen such that

$$R_{P1} + R_{e,h} \gg R_{sense} \| (R_{P2} + Z_{\text{in}})$$

$$R_{P1} + R_{e,l} \ll R_{sense} \| (R_{P2} + Z_{\text{in}})$$

These conditions will ensure that V_{sense} R_{P1} and R_{P2} are current-limiting protection resistors, and their values are chosen so that, should either sensor electrode be directly shorted to either V_{ref} or ground, the current passing through the electrode will be limited to a reasonably small value. For this design, a short-circuit current maximum of 5mA was chosen, giving $R_{P1} = R_{P2} = 1\ k\Omega$. Having chosen values for R_{P1} & R_{P2}, R_{sense} can now be chosen such that $V_{\text{sense}} \ll V_{\text{th}}$ when $R_e = R_{e,h}$ and $V_{\text{sense}} \gg V_{\text{th}}$ when $R_e = R_{e,l}$. For this design $R_{\text{sense}} = 20\ k\Omega$. By solving (5.1) for $V_{\text{sense}} = 1.25V$, the threshold resistance $R_{e,\text{th}}$ where the detected logical signal transitions from low to high can be found to be 56.7 $k\Omega$. Thus, when the resistance R_e drops below

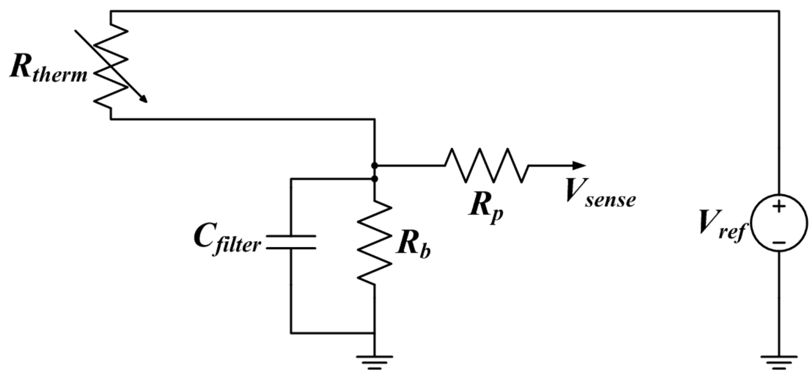


Fig. 5.5: Thermistor temperature sensor circuit diagram

$R_{e,\mathrm{th}}$, the MCU input will transition from low to high. Finally a capacitor, C_{filter} is connected across R_{sense} to provide a shunt path to ground for any high-frequency noise induced on the measurement wiring from the MCB to the sensor location.

5.2.1.2 Temperature Sensor Inputs

The temperature sensors are used to provide temperature feedback from four key points in the thermal system:

- Ambient air temperature
- Fluid heat exchanger temperature
- Air heat exchanger temperature
- TBFPRA ground plane temperature

These temperature measurements are used to provide feedback for the temperature control algorithm running on the MCU and to ensure that the thermal limits of the TEC are not exceeded. In this design, negative temperature coefficient (NTC) thermistors were chosen for the sensor elements due to their low cost, good accuracy over the temperature range of interest, and the simplicity of the measurement electronics. The measurement circuit used for each thermistor is shown in Fig. 5.5. V_{ref}

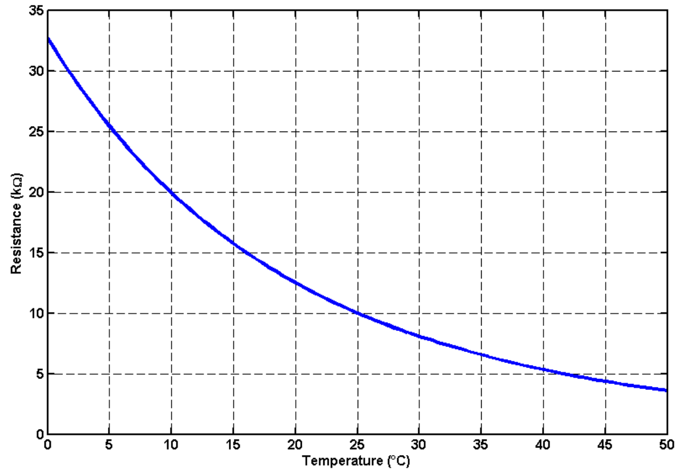


Fig. 5.6: Resistance vs. temperature for Vishay 01M1002KF NTC thermistor

is supplied from a Microchip Technology MCP1541 4.096V precision semiconductor reference source [54]. In this circuit configuration, the thermistor is the top resistor in a resistive voltage divider circuit with a bottom resistor value R_b. The output voltage of this circuit, V_{sense} is fed through a protective current limiting resistor R_p into a high-impedance analog-to-digital converter (ADC) input on the MCU. The ADC converts the sensed voltage into a 10-bit digital value (word) where

$$ADC\ Word = 1023 \times \frac{V_{\text{sense}}}{V_{\text{ref}}} \tag{5.2}$$

This converted data word is then used as the index of a lookup table (LUT) which converts it into a temperature value, which is then passed on to the rest of the firmware code.

The resistive voltage divider circuit shown in Fig. 5.5 is used to help linearize the relationship between V_{sense} and the measured temperature. Fig. 5.6 shows a plot of the relationship between the resistance and temperature of the thermistor used in this work. The resistance behavior of an NTC thermistor with respect to temperature can be described by the Steinhart-Hart equation [55]

$$\frac{1}{T} = A + B \ln R + C (\ln R)^3 \tag{5.3}$$

where R is the resistance of the thermistor (in ohms) and T is the temperature (in Kelvin). A, B, and C are constants derived from a set of measured resistance/temperature points, and are typically provided by the thermistor manufacturer. The inverse of (5.3) can be expressed as

$$R = e^{\left(\sqrt[3]{x-y} - \sqrt[3]{x+y}\right)} \tag{5.4}$$

where

$$y = \frac{A - \frac{1}{T}}{2C} \qquad\qquad x = \sqrt{\left(\frac{B}{3C}\right)^3 + y^2}$$

The sense voltage at the ADC input in the resistive divider circuit is expressed as

$$\begin{aligned} \frac{V_{\text{sense}}}{V_{\text{ref}}} &= \frac{R_{\text{therm}}}{R_{\text{therm}} + R_b} \\ &= \frac{1}{1 + \frac{R_b}{R_{\text{therm}}}} \end{aligned} \tag{5.5}$$

because the ADC input in series with R_p exhibits a large input impedance relative to R_b, so it produces a negligible loading effect on the voltage divider. Substituting (5.4) into (5.5), then substituting that result into (5.2), one can find an expression relating the measured temperature T to the ADC word value of V_{sense}. Fig. 5.7 shows a plot of this function with respect to temperature. The resistance R_b was chosen to be equal to the thermistor's resistance at 25°C. As can be seen, the function provides

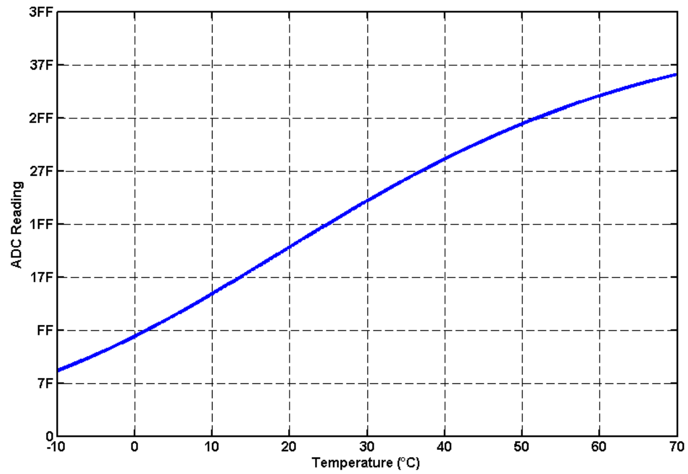


Fig. 5.7: ADC word (hexadecimal) vs. temperature for implemented thermistor circuit

a roughly linear relationship from 0°C–40°C. With the 10-bit ADC on the MCU, this linearization & LUT approach is able to provide 0.1°C per LSB precision over the 0°C–40°C range, and 0.2°C precision from -22°C–67°C.

The LUT approach was chosen for conversion from ADC reading to temperature as a time/memory tradeoff to ensure that the conversion of each ADC reading takes a predictably short amount of time on the MCU. A direct implementation of (5.3) in C code on the MCU would have necessitated a significant amount of processing overhead—particularly if implemented in floating point arithmetic—on the 8-bit processor. This was a particular concern as the firmware performs several dozen individual temperature measurements per second, and those temperature measurements are used as inputs to a delay-sensitive PID control loop. Because the MCU has an ample amount of flash storage, the LUT approach was deemed the best approach for temperature conversion.

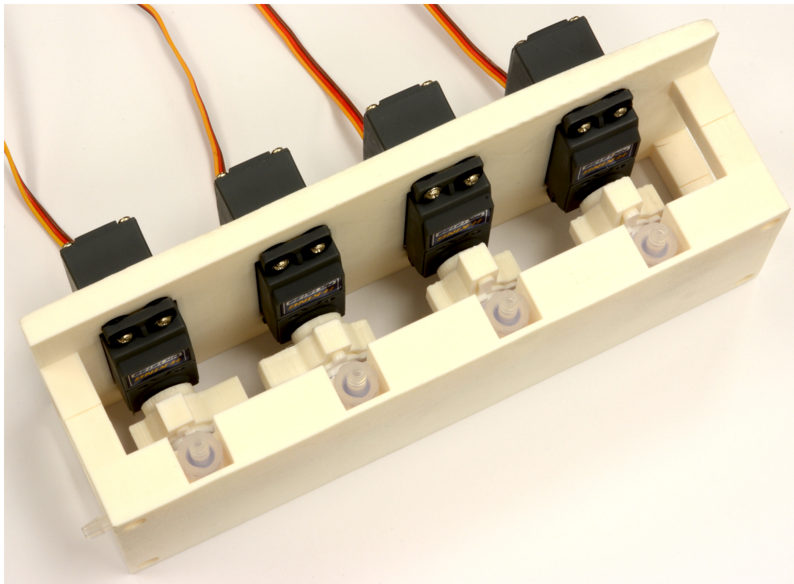


Fig. 5.8: 4x 3-way servo-actuated fluidic valve network

5.2.2 Reconfiguration Control Mechanisms

5.2.2.1 Valve Controls

As shown in Fig. 3.10 on page 59, a set of 8 3-way valves are used to direct the flow of dielectric fluids through the 8 COSMIX elements on the TBPFRA element. To control the position of these valves, a set of servo motors of the type used in radio-controlled hobby models are used. Each servo motor connects to the MCB via a 3-wire interface that carries 5VDC power & ground, and a pulse-width modulated 5V square wave control signal. The PWM signal runs on a 200Hz clock (5ms period), with a positive pulse width that varies from 0.9ms–1.6ms. These pulse widths correspond, respectively, to the far counterclockwise and far clockwise positions of the servo output shaft. When the servos are un-powered they maintain their last commanded position. Since the prototype COSMIX network will require several seconds to pump dielectric fluid through each COSMIX, the valves will spend most of the reconfiguration time stationary. Thus, to simplify the control output design, a single control signal output is multiplexed to all 8 servos, and the 5V power supply to each servo is switched to follow the control signal. This approach requires 1 PWM control signal, 3 address lines, and 1 enable line for a total of 5 control lines from the MCU, as opposed to a parallel control scheme which would require 8 individual PWM control lines. A mount for each group of 4 valves was designed in Dassault Systems' *Solidworks* 3D CAD software and fabricated with a Makerbot Industries *Replicator 2* 3D plastic printer. Fig. 5.8 shows an example of one such servo-actuated 4-valve network.

5.2.2.2 Motor & TEC Controls

To control the pump motors & TEC, a set of five STMicroelectronics VNH5019A-E H-Bridge ICs are used to pulse-width modulate the power supplied. By varying the duty cycle of a 7.8kHz PWM signal generated by the MCU, the average current through the motors (or TEC) can be controlled. By toggling the polarity of the output voltage, the direction of the motors (or the direction of heat flow through the TEC) can be reversed. Fig. 5.9 shows a schematic of an H-Bridge controlling the current through a TEC. Note that the diode chain in the center of the figure represents the chain of metal-semiconductor-metal junctions in the TEC, and is capable of passing current bi-directionally. A PWM control signal is input at the arrows. When the left side is held low and the right side pulsed high, current flows through the upper right MOSFET, through the TEC from right to left, and through the lower left MOSFET. Likewise, when the right side is held low and the left side pulsed high, current flows through the upper left MOSFET, left to right through the TEC, and through the lower right MOSFET.

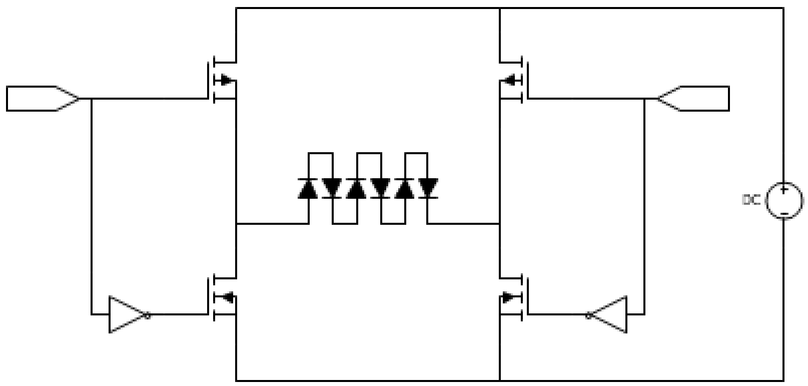


Fig. 5.9: H-bridge circuit controlling current through a TEC

5.2.2.3 Analog Phase Shifter Controls

As one of the proposed functions of the multifunctional array is electronic beam steering, a method of controlling the phase of the RF excitation supplied to the antenna elements is needed. For this implementation, a commercially available MMIC analog phase shifter is used as the phase control element: the Hittite Microwave HMC928LP5E. Fig. 5.10 shows one of the evaluation boards used in the array, with the MMIC mounted in the center. These phase shifter modules are the same as those used in section 3.4.1 to provide a variable reactive load in an open-circuit-terminated configuration.

The HMC928 evaluation board provides SMA end-launch connectors for the RF input & output, as well as the analog DC bias line. As mentioned in section 3.4.1, the HMC928 exhibits an electrical length of roughly 450° across its operating frequency range of 2–4GHz with 0V applied to the bias line. As the DC bias voltage is increased, the apparent electrical length of the HMC928 decreases monotonically up to a maximum bias voltage of 12V, at which point the MMIC's electrical length is reduced to nearly 0°. The relationship between the progressive phase shift and applied

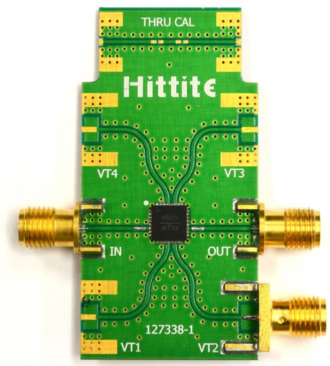


Fig. 5.10: Hittite Microwave HMC928LP5E 2–4GHz 0–450° analog phase shifter

bias voltage for the HMC928 can be modeled as a 3rd-order polynomial fit: [56]

$$V_{\text{bias}} = 2 \times 10^{-9}\theta^3 + 3 \times 10^{-5}\theta^2 + 1.18 \times 10^{-2}\theta + 2.16 \times 10^{-2} \tag{5.6}$$

where θ is the desired phase shift in degrees. This function is programmed into the MCB firmware and is used in the phased array beamsteering algorithm to calculate the bias voltages for each antenna element's phase shifter.

To supply a controllable bias voltage to each of the phase shifters in the array a phase shifter control card—designed by colleague Jeffrey Jensen to control a set of identical phase shifters for similar phased array applications—was used. Fig. 5.11 shows a block diagram overview of the control card. The card's design centers around an Analog Devices AD5668 8-output, 16-bit digital-to-analog converter (DAC) IC. The DAC IC is controlled by the MCU over an SPI synchronous serial interface. The DAC provides a set of 0–5VDC output voltages, which are amplified by a pair of Texas Instruments OPA4705 quad CMOS op-amp ICs to generate a set of 0–12V DC bias signals. These signals are then supplied to the HMC928 boards over a set of SMA-connectorized coaxial cables.

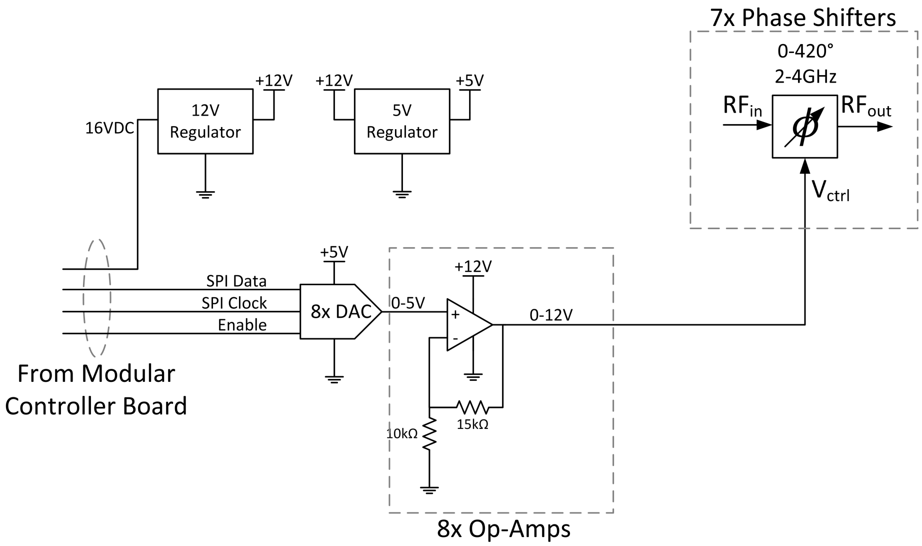


Fig. 5.11: Block diagram of phase shifter control DAC card

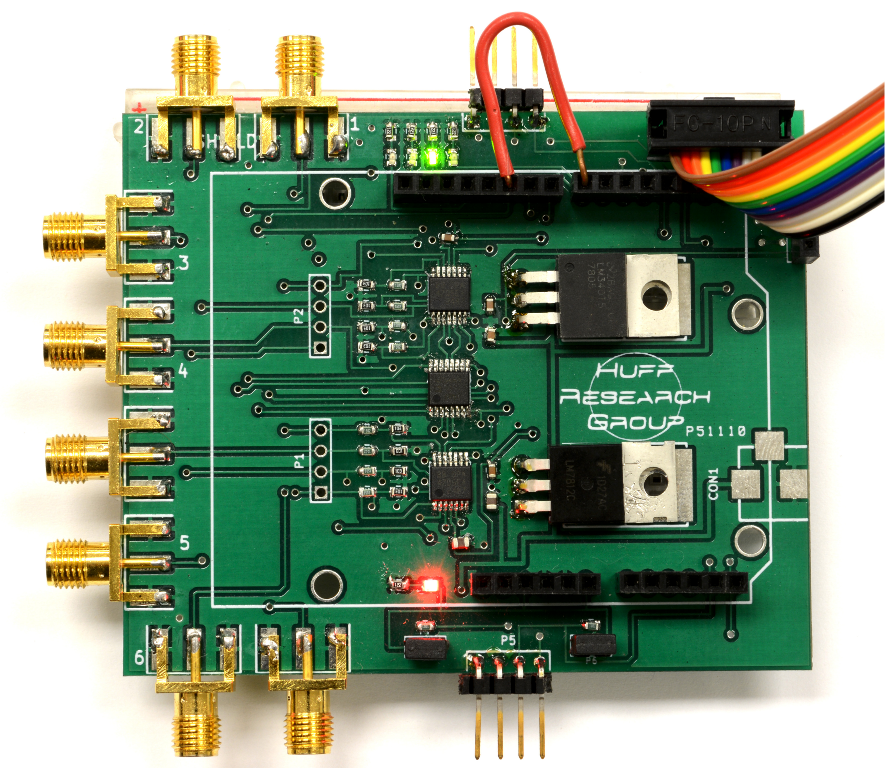


Fig. 5.12: Phase shifter control DAC card

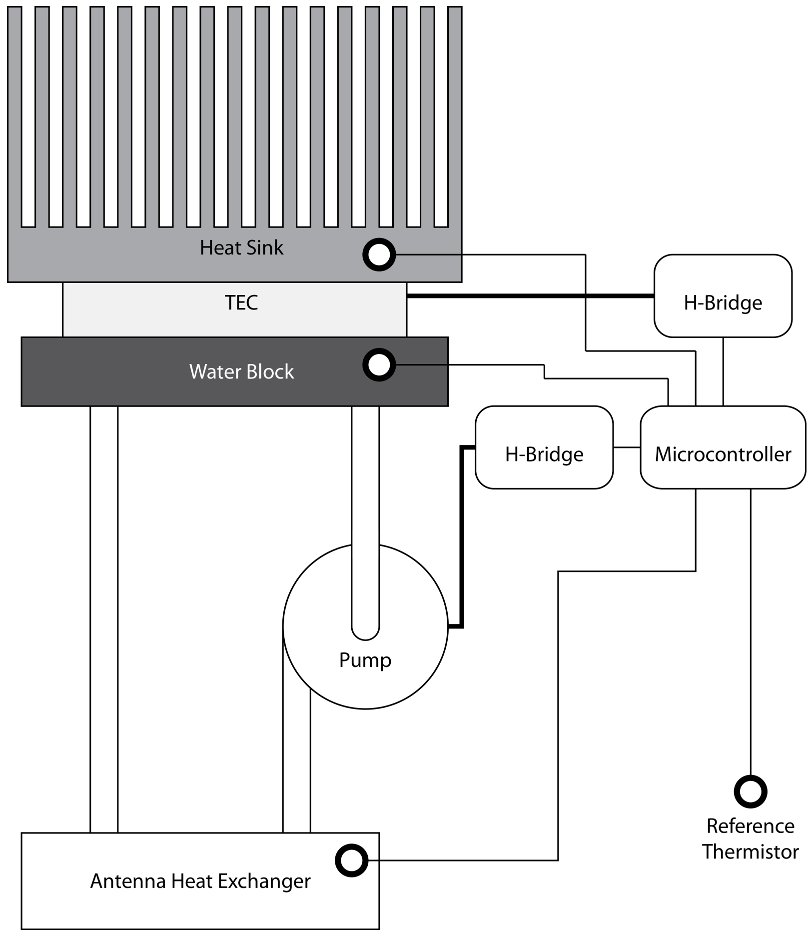


Fig. 5.13: Block diagram of antenna element thermoregulation system

During the design of the control card, the author collaborated with Mr. Jensen to develop a common 8-pin power & data connector pinout. This ensured the interoperability of the MCB with the phase shifter control card. Fig. 5.12 shows an image of the assembled phase shifter control card. The multicolored ribbon cable connected in the upper right-hand corner is the data & power interface to the MCB.

5.2.2.4 Thermoregulation System

In order to control the temperature of each antenna element, a closed-loop thermoregulation system was designed to interface to the MCB to provide the necessary

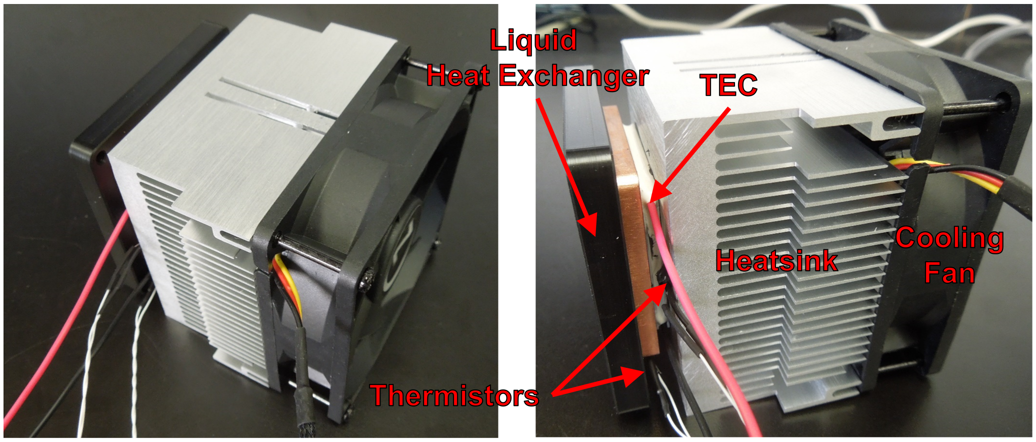


Fig. 5.14: Assembled Peltier TEC heat pump

bidirectional heat pumping. Fig. 5.13 shows a schematic overview of the designed thermoregulation system. A Peltier TEC sandwiched between an air-cooled finned aluminum heatsink and a copper liquid heat exchanger forms the heat pump module. A brushless DC fountain pump provides motive force to a heat exchange fluid, which cycles between the heat pump and a heat exchanger on the backplane of the antenna element. A set of H-bridges allow the MCB to control the power supplied to both the pump and TEC. Thermistor temperature sensors located on both sides of the TEC heat pump and on the antenna element provide temperature feedback to the discrete PID controller running on the MCU.

An assembled TEC heat pump is shown in Fig. 5.14. The assembled module consists of a comercially available CPU cooler (Rosewill RCX-Z80-AL) and fan assembly, a CUI, Inc CP85438 TEC module, and a copper CPU cooling waterblock. The heat pump in Fig. 5.14 was assembled using a thermally-conductive epoxy (MG Chemicals 8329TCM-6ML). After testing this assembly method, it was observed that the thermal cycling experienced during heat pump operation resulted in a failure of the epoxy bond. Thus, the full set of heat pumps were assembled by applying thermal silicone grease between the heat sinks and TEC and using plastic zip-ties to apply compressive force to hold the heat pump assembly together.

5.2.3 Processing, Communication, & Firmware

5.2.3.1 Microcontroller & Wireless Radio

The key component of the MCB is the microcontroller (MCU), which runs a custom firmware that monitors the sensor inputs, runs a PID temperature control algorithm, and listens for commands on an asynchronous serial interface. For this design, a *Teensy++ 2.0 USB Development Board* supplied by PJRC.com [57] was chosen as the MCU platform. The *Teensy++ 2.0* is based on an Atmel AVR AT90USB1286 8-bit MCU running on a 16 MHz clock. The *Teensy++ 2.0* runs a custom bootloader which allows compiled main firmware to be easily uploaded from an open-source loader application via an onboard USB connection.

The wireless data link from the MCB to the array control server is provided by a Digi International *XBee WiFi S6* wireless radio module (XBee). The XBee interfaces with the MCU via a three-wire asynchronous serial connection running at 9600 baud. The XBee module was chosen because, once it is properly configured, it provides a transparent interface between the serial link and a TCP socket over WiFi. The XBee module provides a complete 802.11b/g WiFi protocol implementation, including WPA encryption, and a full TCP/IP networking stack. This removes the need for the MCU firmware to implement any high-level networking protocols.

5.2.3.2 Firmware

The firmware consists of a boot-up routine, main loop, and hardware timer-triggered interrupt routine. The boot-up routine, which is run immediately after power-on, configures the on-board MCU peripherals (ADC, PWM outputs, hardware timers, asynchronous serial port, and digital input pins) and the sends the necessary configuration commands to the XBee. The main loop reads characters from the serial port, responds to recognized commands, and runs the PID temperature control algorithm when triggered by the hardware timer. The hardware timer interrupt is configured to run at 1kHz, and manages the precise triggering of the PID routine and time interval measurements. Fig. 5.15 shows a block diagram representation of the main firmware program flow.

On power-up, the MCU configures the XBee to connect to a the array control WiFi network using a preprogrammed SSID & encryption passphrase. The firmware then waits and checks to ensure that the XBee properly connects to the network. Once the XBee reports that it has connected, the firmware sends a data packet containing its serial number to a the control server's IP address, which is preprogrammed into the firmware.

After sending the announcement packet to the control server, the firmware initializes the command parsing routine and begins listening for commands from the control server. A reference for the commands and syntax supported by the MCB firmware is listed in Appendix C. In general, every command string consists of a root command, optionally followed by a space and a variable number of space-delimited arguments. A valid command string is terminated by a carriage return character (Hexadecimal byte value: `17`). When the parsing routine receives a `17` it extracts the root command from its receive buffer and begins comparing it to the list of known commands. If the received command matches a known command, the associated handler function for that command is called to execute the appropriate action and generate a response. If the received command does not match any known command, an `ERROR` response is returned. The command handler functions are responsible for interpreting command arguments and directly interfacing with lower-level drivers for the various reconfiguration mechanisms.

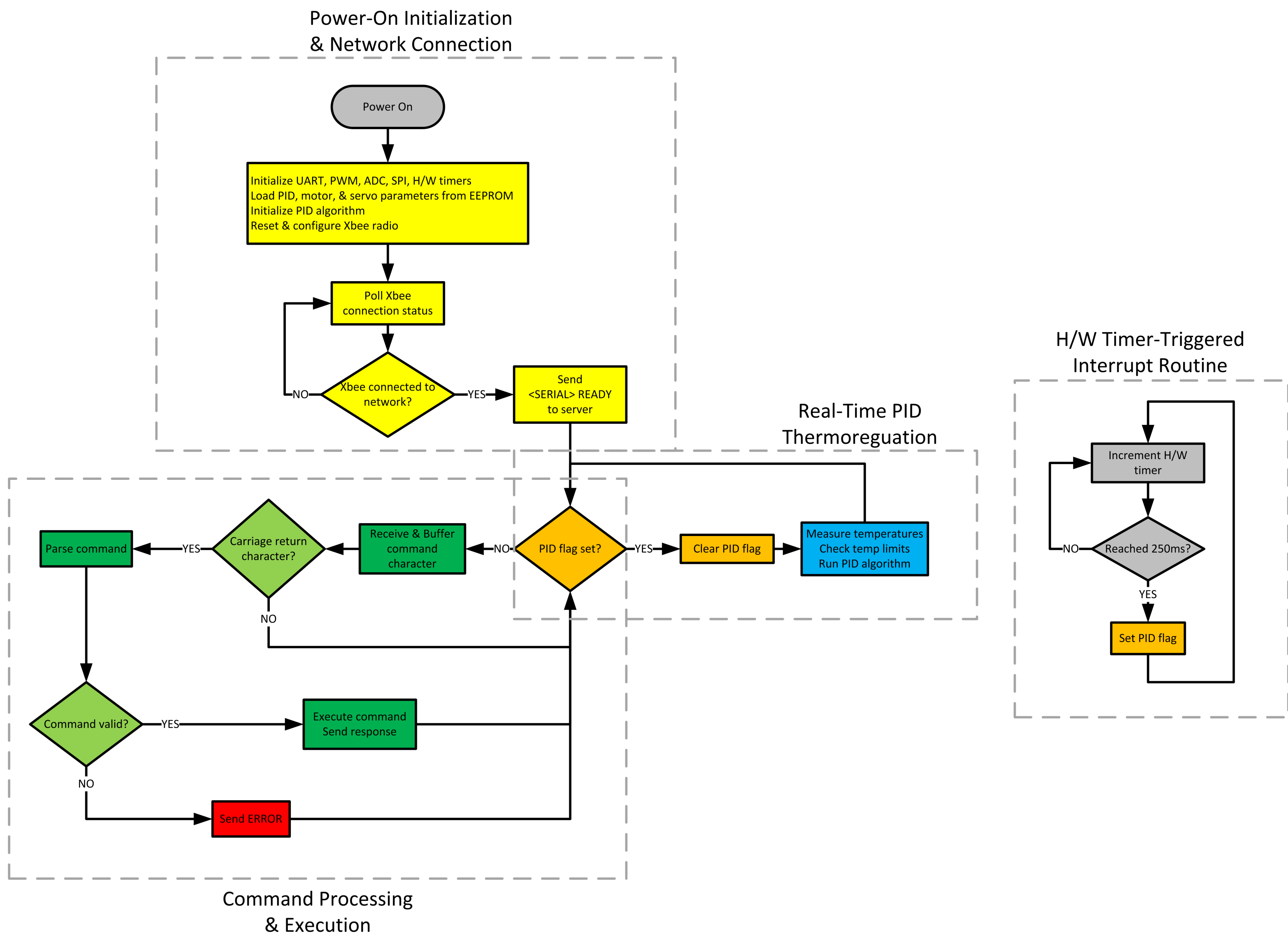


Fig. 5.15: MCU firmware block diagram

The timer-triggered thermoregulation function implements the discrete PID control algorithm used to manage the temperature of the antenna element. The function is triggered by a flag bit set in a hardware timer interrupt function, and runs every 250ms (4 Hz). When the thermoregulation function is called, it first takes ADC readings from every temperature sensor input and converts them to temperature values. It then checks the measured temperatures against a set of low & high limits. Since an open or short circuit on a thermistor would result in an extremely low or high measured temperature value, respectively, this limit check serves the dual purpose of keeping the thermal system within safe operating limits and checking for thermistor faults. If the limit check fails, the function immediately shuts down the TEC and goes into a "SCRAM" mode, from which it must be reset manually. This ensures that the thermal system cannot exceed the thermal limits of the TEC or antenna element and damage itself. Otherwise, if the PID loop is in the "AUTO" mode the function computes a new TEC control output from the PID loop and returns.

Another safety feature of the firmware design is built into its powerup sequence. By default, the MCB firmware initializes all of its control outputs in a quiescent state. In particular, the thermoregulation PID loop is kept off at boot-time, and must be engaged by the array control server every time the MCB boots up. This ensures that the MCB does not start up in an unknown or ill-defined state from which its behavior might be unpredictable.

5.3 Fabrication & PID Algorithm Tuning

A set of printed circuit boards (PCBs) based on the MCB design described in section 5.2 were ordered & fabricated at Advanced Circuits, Inc. [58] The PCBs were assembled in house by the author. Fig. 5.16 shows the fabricated & assembled board design. Along the top of the board are the five H-bridge motor & TEC driver

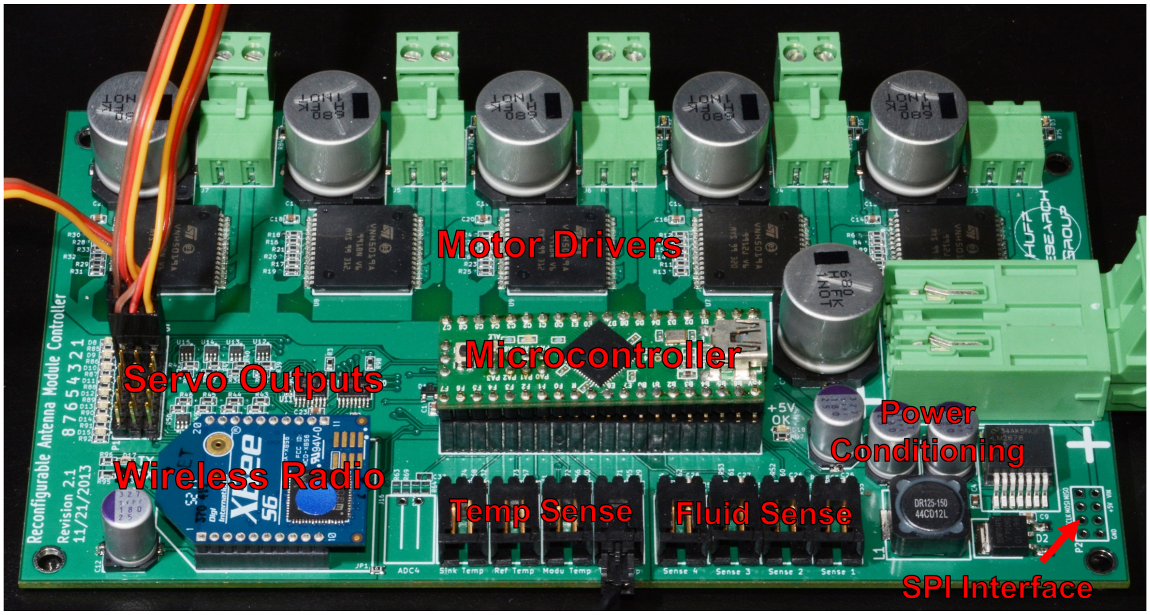
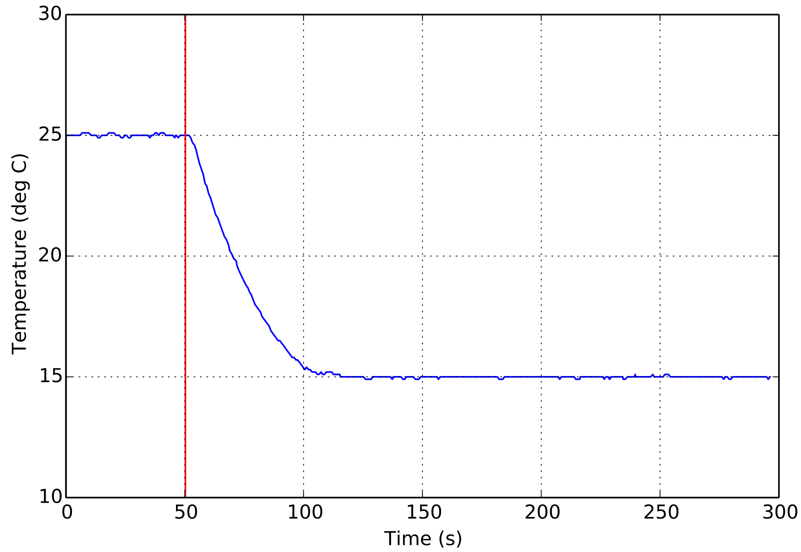


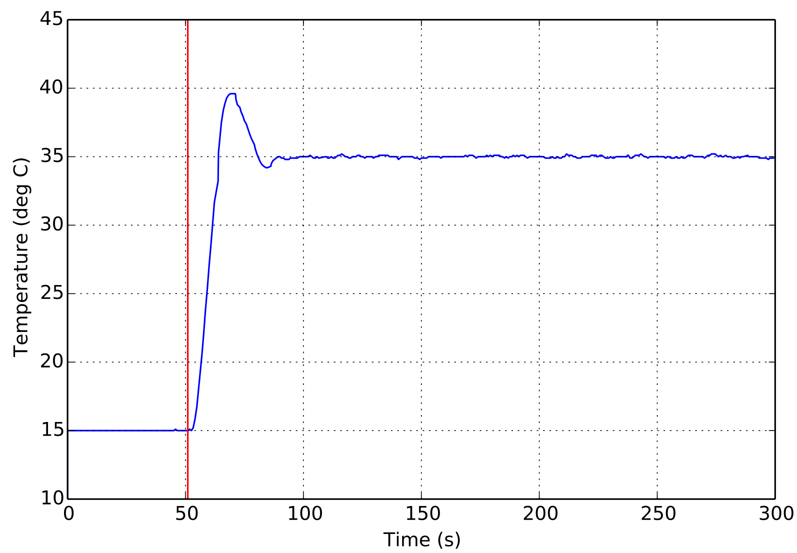
Fig. 5.16: Fabricated modular controller board

circuits, with their associated energy storage capacitors & output connectors. On the left side is the power switching & control multiplexing circuit for the servo valves. In the figure, three such valves are connected to the output pin header. Below the servo outputs is the XBee WiFi radio module. Along the bottom of the board are the locking connectors for the temperature & fluid sensor inputs. In the bottom-right corner is the 5V, 5A buck converter (and its associated filter capacitors & inductor) that provides power to the board and servo motors. Also on the bottom-right corner is the 8-pin SPI & power interface header to connect the MCB to the phase shifter control card. Along the right edge is the main power input connector & filter capacitor. Finally, in the center of the board is the MCU module.

The PID temperature control algorithm was tuned and tested by running a series of step tests, wherein the temperature setpoint was changed from one value to another. Tuning was accomplished by running the control algorithm in open-loop mode, instigating a step change in the controller's output, and measuring the temperature response of the thermal system. From the measured response, the controller gains for the PID controller were calculated using the Cohen-Coon tuning method as described in section 2.6. After calculating the appropriate P-, I-, & D-gain parameters for the thermoregulation loop and loading them into the EEPROM of the MCB, the tuned controller was tested by issuing a set of step changes in the temperature setpoint. Fig. 5.17 shows the results of these step tests on the tuned PID controller. Note that these tests were performed without fluid in the heat exchange loop. Fig. 5.17b in particular shows the exact response expected from the Cohen-Coon tuning method. The first overshoot is roughly 4°C, and the following undershoot is roughly 1°C, exactly a quarter-amplitude decay response.



(a) Setpoint step from 25°C to 15°C



(b) Setpoint step from 15°C to 35°C

Fig. 5.17: Tuned PID temperature controller test results

6. ARRAY CONTROL NETWORK & SERVER

6.1 Design Goals

The purpose of the array control network is to provide dynamic, wireless control of the reconfiguration mechanisms on individual array modules. Each module in the array consists of a MCB, antenna (EPRA/TBPFRA) element, TEC heat pump, fluid pumps, and valve network. One of the primary design objectives for the array control network was to allow dynamic reconfiguration of the array—known as *plug & play*. In particular, the array control system was designed so that individual modules could be added or removed from the array in real time without requiring the entire array to be reset or have its operation interrupted in any way. Such a plug & play design allows the array control network to be used for dynamic array reconfiguration experiments, and also provides fault-tolerance by ensuring continued operation of the array as a system despite failures on the individual module level. Additionally, the control network is designed such that multiple user-interface (UI) clients can communicate with and control the array simultaneously.

6.2 Array Network Design

In order to achieve these goals, the array control network was designed in a star-topology. An overview of the control network layout is shown in Fig. 6.1. A central array control server forms the hub of the star, and interfaces with other nodes (UI clients & antenna modules) via an infrastructure-mode WiFi wireless network. The array control server is the central node of the control network, and provides the interface between UI clients and the individual modules in the array. Thus, modules only ever communicate directly with the control server during normal operation. Similarly,

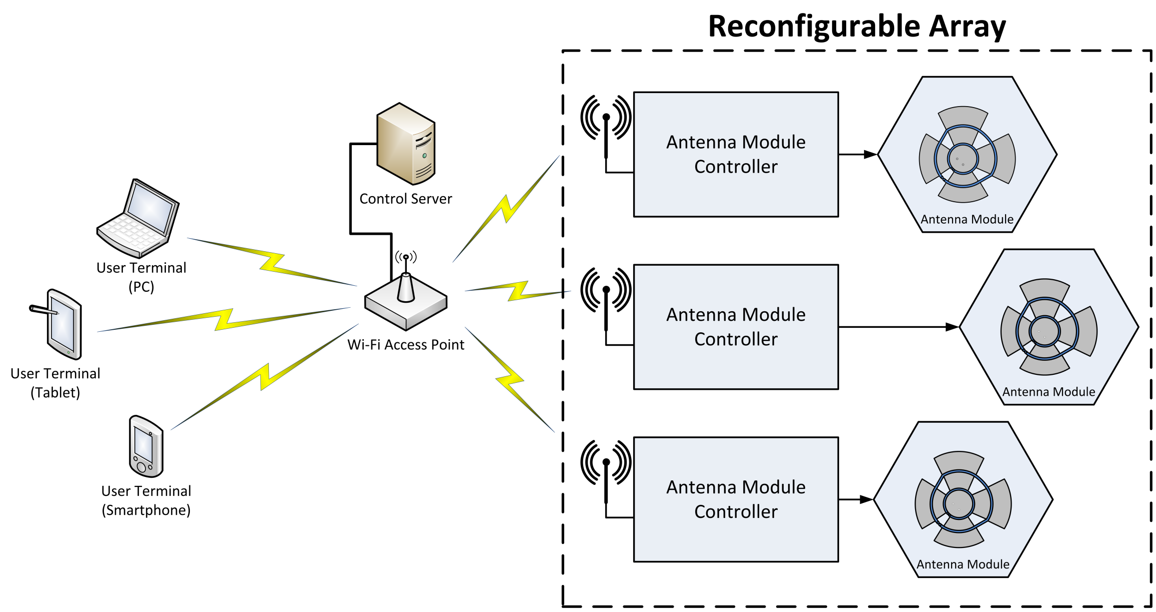


Fig. 6.1: Array control network structure

UI clients also only interface with the control server. The control server performs array management, module tracking & identification, and monitoring tasks. The UI client can then request data about the array from or push reconfiguration commands to the control server, which dispatches the relevant module-level commands to the modules in the array. The antenna module controllers (MCB & associated reconfiguration peripherals) then process these commands and actuate the antenna elements to achieve the commanded reconfiguration or retrieve the requested measurement data.

6.3 Control Server Implementation

The control server application was implemented in the Python programming language. Python was chosen for several reasons. First, it is an interpreted language, so prototype iterations are significantly faster as there is no compilation step required. Second, Python is a high-level programming language which provides a large number of libraries to facilitate activities like network & database access, further easing development. An overview of the control server application architecture follows.

6.3.1 Application Structure & Data Flow

Fig. 6.2 shows a graphical overview of the architecture of the control server. The application is written in a multithreaded style. This is because a considerable amount of the server's functionality concerns network communication. As network read/write operations have inherently variable latency (particularly so with TCP socket-based network communication), the multithreaded approach ensures that delays in network communication operations with one particular client do not result in unresponsiveness to other clients. (Note that the CPython interpreter implements a *Global Interpreter Lock* which precludes the use of true native multithreading, but this is not relevant for the network operations of the array control server.)

Fundamentally, the array control server application is composed of three main components, each of which run in separate threads:

- the *main server logic*, which is responsible for starting the other components, mediating communication between UI clients and the array, and implementing the thermal signaling algorithm explored in section 7.1,
- the *UI client server*, which handles connection requests from UI clients and interfaces them with the main server logic, and
- the *module handlers*, which interface with individual antenna modules, gather module tracking information, and distribute module-level commands

On startup, the server connects to a local SQLite database that stores state & identification information for the modules on the network, and opens two listening

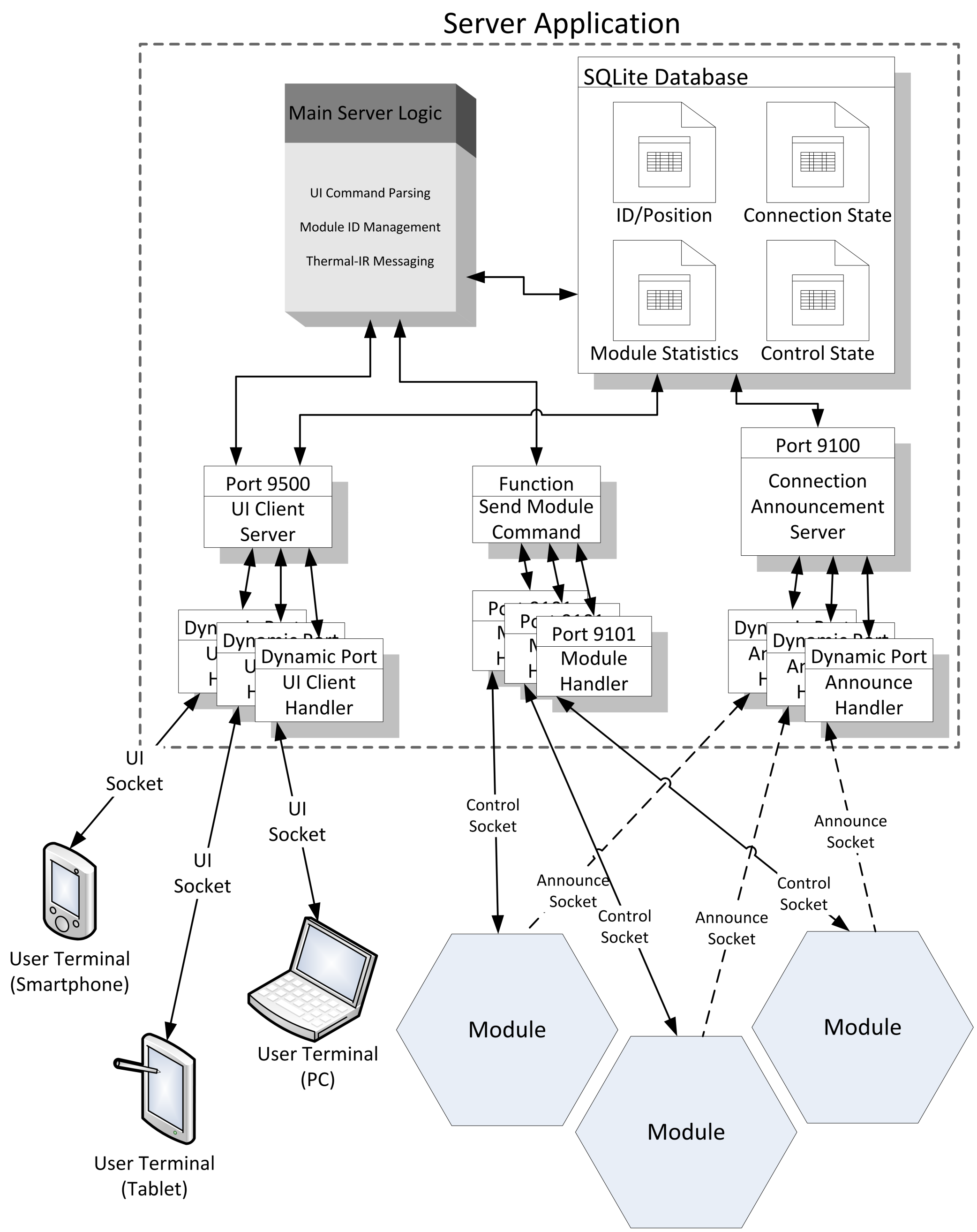


Fig. 6.2: Block diagram of server application design

server sockets: a UI client server on port 9500, and a module connection announcement server on port 9100. In operation, the control server and UI client communicate using a purely client-server architecture, where the UI client application acts as a client and requests array control services from the control server. In the communication path from control server to modules in the array, however, the modules only act as clients during their boot-up & initial network connection. During steady-state operation each module acts individually as a server, and the control server application connects to each module as a client, sends it a command, and retrieves its response.

6.3.2 UI Client Server

The UI client server listens for UI clients to connect on port 9500. When a UI client application connects, the UI client server spawns a UI client handler in a new thread and passes the connected socket to the client handler. The UI client handler then communicates with the UI client application. First, a status summary is sent to the client application which indicates the client is connected and gives a summary of the state of the array as shown below.

```
Connected to Array Server v.0.1
Modules Connected: 1
SERIAL:         ID:     IP ADDRESS:
23A71F96CD      0       10.0.0.167
```

Next, the client handler waits for a command from the UI client application. When the UI client sends a command, the command and client socket ID are placed into an RX queue shared by all client handlers and the main server logic. This RX queue passes the command to the main server logic, where it is parsed and processed. The server logic then executes the appropriate action and pushes a response, tagged with the client socket ID, into a TX queue. The client handlers watch this TX queue, and when a message with a client handler's respective client socket ID appears the message is pulled out by the client handler and transmitted to the UI client. Thus, the control server can interface with multiple UI clients on a first-come first-served basis.

Table 6.1: Command structure for UI client → control server communication

Command Scope	**;**	**Variable Command Format**		
`array`	;	`<MODULE COMMAND>`		
`module`	;	ID List (comma separated)	;	`<MODULE COMMAND>`
`server`	;	`<SERVER COMMAND>`		
`irmsg`	;	`<MESSAGE TEXT>`		Inter-Character Delay (s)

6.3.3 Module Connection Announcement Server

The module connection announcement server ("announce server") listens on port 9100. The purpose of the announce server is to listen for module connection announcements. Whenever a module's MCB is powered on, it boots up and tries to connect to the array control network. Once the module's on-board WiFi radio indicates it has successfully connected to the network, the MCB connects to the announce server on port 9100 and sends a string of the form `<SERIAL NO.> READY` where `<SERIAL NO.>` is the media access control (MAC) address of the WiFi radio module. Since MAC addresses are (theoretically) unique for all IEEE 802-compatible networking devices, they are used as a unique serial number for each module on the network.

As with the UI client server, a connection request from a module to the announce server spawns an announce handler in a new thread. Thus, the server can handle a large number of simultaneous connection announcements as when many modules are powered on simultaneously. When the announce handler receives an announcement packet, it validates the packet format. If the packet is correctly formatted, the IP address and serial number of the module are recorded, and the module's entry in the module state database is updated with the current time, serial, and IP address. Once the database is updated, the announcement socket is closed. Thus, the announcement server & announce socket is only used once, immediately after each module boots up.

6.3.4 Module Command Handler

Whenever a UI client command or server function requires that a command be dispatched to one or more connected modules, the command is routed through the module command handler. The main server logic passes the command to be sent and a list of module IDs to the module handler, which spawns a handler thread for each module to be addressed. The individual handler threads then check the module state database to verify that the module is connected and pull the module's IP address. The handler thread then opens a client socket to the module on port 9101 and sends the command to the module. Finally, the handler thread collects the module's response and returns it, a status code, and the module's ID back to the module command handler. The command handler collects the status codes & responses from all the handler threads and returns them to the server logic. Finally, the server logic returns the ID list, status codes, and responses from the modules to the UI client application via the TX queue.

6.3.5 Network State Management

As previously described, the array control server keeps track of connected antenna modules in a SQLite database. UI clients use unique *module IDs* to reference single or groups of modules. These module IDs are (typically) sequential integers from

0 to (in the case of the 7-element array) 6. These ID numbers are dynamically configurable from the UI client, and provide an abstraction layer which dereferences the UI client's identification of antenna modules from the physical antenna module hardware (and its associated unique 12 digit hardware serial number). Before a module can be controlled from the UI client, it must first be assigned an ID number. The control server provides a set of commands to accomplish this. The UI client can command the server to automatically assign IDs to all modules in the array, in which case the server assigns IDs sequentially starting at 0 to all connected modules in the order the modules first connected to the control network. The UI client can also directly assign a specific ID number to a specific module by specifying the module's hardware serial number.

Since the control server does not keep a TCP socket continuously open to each module (and in fact cannot, due to firmware limitations of the XBee WiFi modules), the only way for the server to determine if a module has disconnected from the array is by polling all modules in its database. The module handler function provides feedback to the database on the module's connection status. Whenever a command—polling or otherwise—is queued to be sent to a module the module handler thread attempts to open a TCP socket to that module's IP address. If the socket connection request times out after 3 connection retries, the handler thread returns an error status code and marks the module as disconnected in the database. Furthermore, since the MCB firmware generates a response to all module commands—including corrupt or improperly formatted ones—if the module handler fails to receive a command response during the timeout period it will also mark the module as disconnected.

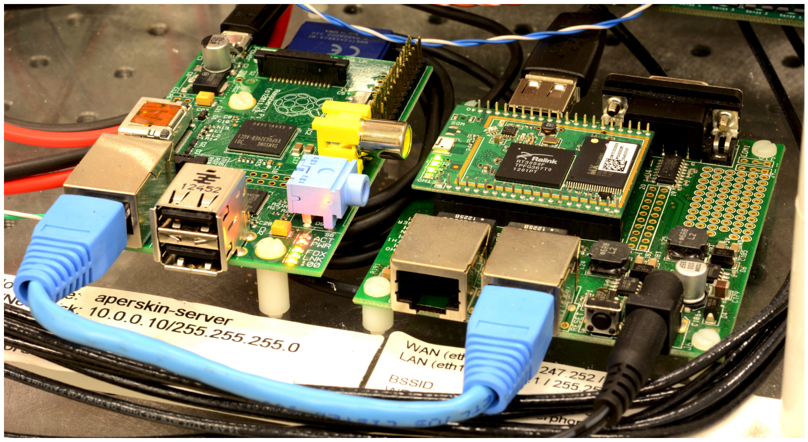


Fig. 6.3: *Raspberry Pi* array control server (left) and *Carambola* WiFi router mounted on array testbed

6.4 Server & Network Hardware

During experimental operation, the array control server application was run on a Raspberry Pi Foundation *Raspberry Pi* single-board computer. The Raspberry Pi is a low-cost, low-power computer based on a Broadcom BCM2835 ARM system-on-a-chip (SoC). The Raspberry Pi's operating system is the *Raspbian* Linux distribution, a fork of the popular *Debian* Linux distribution. Raspbian comes preinstalled with many of the software packages necessary to run the array control server software, including a Python 2.7 environment and an SSH server for remote administrative access over the network. The network routing, DHCP service, and wireless network connection for the antenna modules & UI client was provided by an 8devices *Carambola* WiFi development board. The Carambola is a networking development board based on a Ralink RT3050 SoC with built-in 802.11b/g/n WiFi radio and dual 802.3 100BASE-TX wired Ethernet interfaces. It runs another custom Linux operating system: *OpenWRT*. OpenWRT is a lightweight Linux OS targeted at low-power wireless access point/router hardware like the Carambola. The default distribution of OpenWRT on the Carambola comes preinstalled with a variety of programs which allow it to provide the same networking services as a consumer-grade wireless router. The Raspberry Pi interfaces with the Carambola via a short wired 100BASE-TX Ethernet connection.

The server computer is configured with a static IP address which is also programmed into each antenna module's firmware. The modules themselves request & receive dynamic IP address assignments from the DHCP server on the Carambola each time they connect to the network—this is why the server logs each module's IP address when it makes its power-on announcement. Using DHCP obviates the need for each module to be pre-programmed with a unique static IP address. Fig. 6.3 shows the array control server computer and wireless router as mounted on the array testbed.

7. FULL MULTIFUNCTIONAL ARRAY SYSTEM EXPERIMENTAL RESULTS

After all of the individual components and subsystems had been successfully tested and their functionality verified, they were integrated into the multifunctional array system. Fig. 7.1 shows the assembled 7-element hexagonal antenna array and its associated control systems. The testbed consists of a multi-layered platform housing the reconfiguration control systems and a mount for the planar multifunctional antenna array. The physical structure is constructed from 1/4-inch thick acrylic (Plexiglas) sheets measuring 17 inches deep by 24 inches wide, with 1/2-inch ID PVC pipe fittings forming the inter-layer connections and array mounting structure. The EPRA elements are mounted in a regular hexagonal grid with 85.1mm center-to-center spacing. The element mounting structure is a 10mm thick sheet of ROHACELL HF structural/RF foam measuring 16 inches square. The ROHACELL sheet is fastened, in turn, to the PVC pipe mounting structure by two polypropylene (PP) nuts which thread onto matching PP screws threaded into the PVC mounting structure. The mounting holes for the screws are spaced 12 inches apart in the ROHACELL sheet, centered about the center antenna element's probe feed. A second set of mounting holes is located above and below the array, allowing it to be rotated about the z-axis to facilitate orthogonal-plane radiation pattern measurements with a common phase-center and without requiring the entire testbed structure to be tipped on its side.

The bottom layer of the structure contains the majority of the array control system: the seven modular controller boards, DC bias tees, array control server board, and WiFi access point. The middle layer contains the servo valve networks

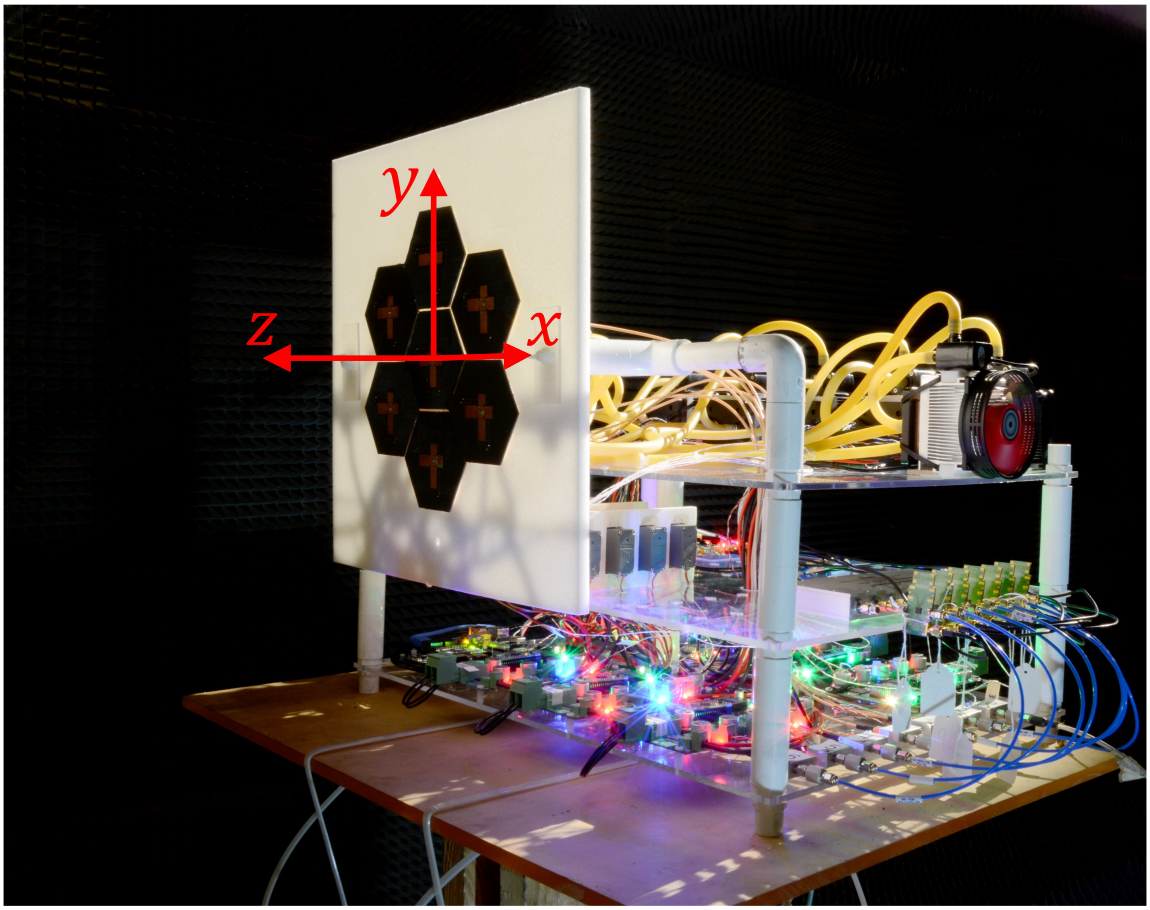


Fig. 7.1: Assembled multifunctional reconfigurable antenna array

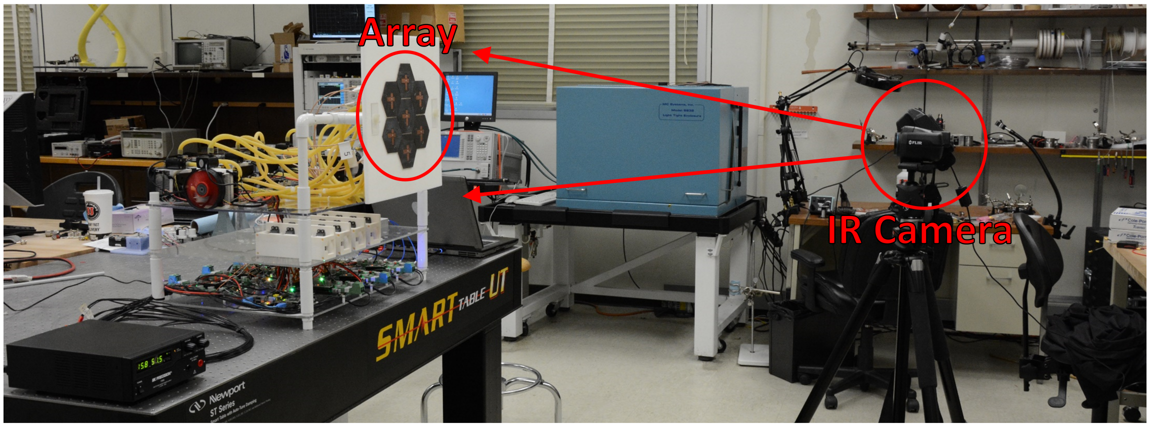
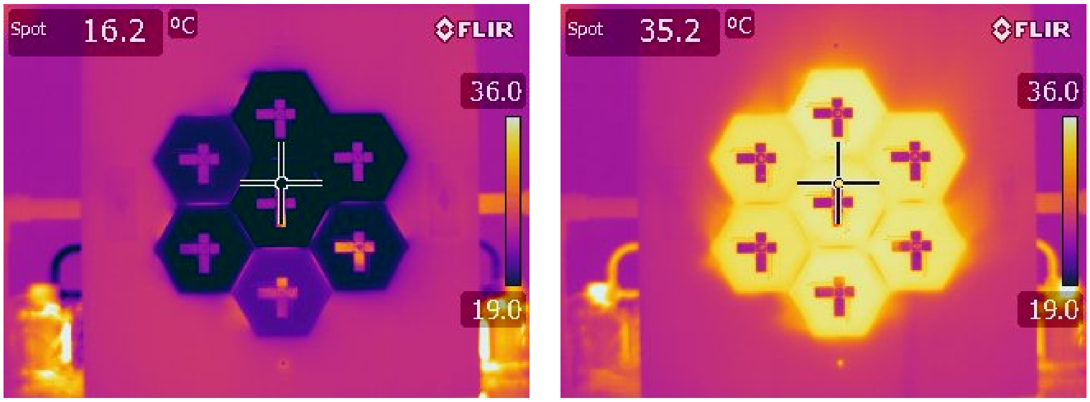


Fig. 7.2: Thermoregulation test experimental setup

for fluid handling, 8-way RF power divider, analog phase shifters, and phase shifter control DAC card. The top layer contains the themoregulation heat pumps for all seven modules, the heat exchange fluid networks & pumps, and power distribution for the heat sink fans. A set of holes at the center of each layer allow the RF feed lines, control, power, and sense wiring to pass between the layers.

7.1 Thermoregulation & Thermal-IR Signaling Experiment

The first set of experiments performed with the multifunctional reconfigurable array were thermoregulation tests to assess the control system's ability to modulate the temperature of the individual EPRA elements. Fig. 7.2 shows the experimental setup for these tests. A FLIR Systems T440 infrared camera with a 320x240 pixel microbolometer was positioned on a camera tripod roughly 1.2m directly in front of the plane of the array, centered about the array's z-axis. Each element's heat transfer fluid loop was primed with roughly 60mL of distilled water. There is some variance (estimated to be roughly 10mL) in the primed volume of each element's fluid loop, as the length of 1/4 inch ID latex tubing between each EPRA's heat exchanger and associated TEC heat pump is different. Nevertheless, the intent of



(a) Setpoint: 15°C　　(b) Setpoint: 35°C

Fig. 7.3: Array-wide thermoregulation

this experiment is to gain a rough sense of the thermal dynamics of this system, so this variance is considered acceptable. All of the thermoregulation tests were performed in a controlled laboratory environment at 25°C ambient air temperature.

7.1.1 Thermoregulation

For the first test, the PID control loop on each EPRA element was engaged and the entire array was commanded to slew to a common temperature. Fig. 7.3 shows the results of the first thermoregulation tests. The thermal images were taken roughly 3 minutes after the setpoint change command was sent to allow each element's temperature to stabilize at the setpoint value. One of the fundamental characteristics of the array's thermoregulation system observed during these tests is the asymmetry in each element's heating & cooling rates. Explicitly, each element is capable of raising its temperature much quicker (corresponding to a higher thermal power when pumping heat into the fluid loop) that it can lower it (a lower thermal power when pumping heat out of the fluid loop). This observation can be expressed as

$$\dot{Q}_{\text{heating,max}} > \dot{Q}_{\text{cooling,max}} \tag{7.1}$$

This observation makes sense when considering the physics at play in the thermoregulation loop, particularly in the Peltier TEC. As discussed in section 2.5, the TEC exploits the Peltier thermoelectric effect to move heat directly with a current flowing through a set of semiconductor-metal junctions. Because the physical TEC module exhibits a non-zero resistance in the metal & semiconductor elements, Joule heating results in the conversion of supplied electrical energy into heat. This *waste heat* is dissipated into both the hot & cold sides of the TEC. On the cold side, this means that the heat pumping mechanism must move both heat from the cold system and waste heat dissipated by Joule heating to the hot side. On the hot side, both the pumped heat from the cold system and waste heat from the TEC must be dissipated such that $\dot{Q}_h \approx \dot{Q}_c + \dot{Q}_{\text{Joule}}$. Thus, the $\dot{Q}$ dissipated on the hot side of the TEC is always larger than $\dot{Q}$ extracted from the cold system by the cold side of the TEC. For the CUI CP85438 Peltier TECs used as the heat pumps in this system, when cooling the elements to 15°C with a 55°C heat sink temperature, this translates to $\dot{Q}_{h,\text{out}} \approx 4.5\dot{Q}_{c,\text{in}}$. This behavior also means that when the thermoregulation system is commanded to slew to a temperature away from that of the ambient environment, the system is capable of properly regulating a in a wider range of temperatures above ambient temperature than it is below ambient temperature, or

$$T_{\text{max}} - T_{\text{amb}} > T_{\text{amb}} - T_{\text{min}} \tag{7.2}$$

7.1.2 Thermal-IR Signaling

The second set of experiments performed with the thermoregulation system explore the novel concept of *thermal-IR signaling*, wherein the temperatures of individual elements in the array are modulated to encode digital information. It is well established that the temperature of an object is related to the quantity and spec-

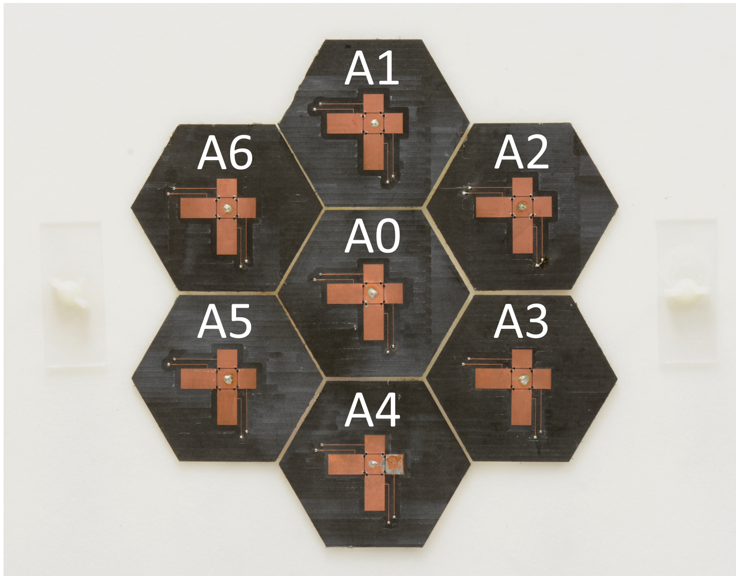


Fig. 7.4: Logical numbering of EPRA elements in the hexagonal array

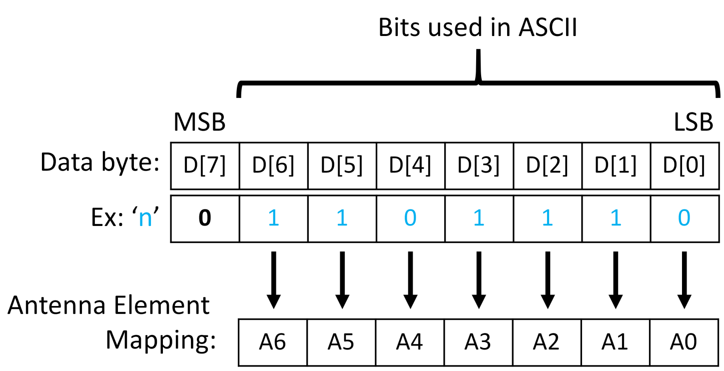


Fig. 7.5: Mapping ASCII-encoded bits to antenna elements

tral density of electromagnetic radiation emitted due to the thermal motion of the charged particles in the object—this is the principle upon which thermal-IR imaging is based. Thus, by modulating the temperature distribution across the plane of the array, the power and spectrum of the thermal radiation emitted by different areas of the array can be modulated. With this system, the array can be used as a raster display device analogous to a visible-light display such as a computer monitor, except with its emissions constrained to the long-wave infrared (thermal-IR) wavelength range. Such a thermally-modulated array could be employed, for example, as a form of covert communication.

Table 7.1: Temperature representation of bit values

Bit Value	**Element Temperature**
0	20°C
1	30°C

To explore this concept, a basic modulation scheme was developed to transmit a string of printed text characters as 7-bit ASCII-encoded symbols, represented by the temperature of the elements in the hexagonal array. The elements in the array were logically numbered according to Fig. 7.4. The bits in each ASCII-encoded symbol are logically mapped to antenna elements such that the least-significant bit (LSB) is mapped to antenna element A0 and the most-significant bit (MSB) is mapped to antenna element A6. The full mapping is shown in Fig. 7.5. Next, a set of temperatures were assigned to represent the respective value of each bit. For the purpose of this experiment, an element temperature above ambient was chosen to represent a binary `1`, and a temperature below ambient to represent a binary `0`. The actual temperatures used are shown in Table 7.1.

The conversion of message characters to element temperature setpoints is handled by the array control server. The UI client sends a command to the server of the form `irmsg;`<Message Text> <Inter-Character Delay> where <Message Text> is a string of printable ASCII characters forming the message to be transmitted and <Inter-Character Delay> is an integer specifying the time delay—in seconds—the server waits between sending temperature setpoint commands for consecutive message characters.

To assess the multifunctional array's performance in thermal signaling, a series of three-character messages were transmitted from the UI client. In the first signaling test, the text `ONR` was transmitted. The resulting thermal signature of the array during each character's signaling time period is shown in Fig. 7.6. This figure shows that the elements' temperature differences are clearly visible in the thermal-IR wavelength range. Fig. 7.6 also shows the corresponding translation of the element temperatures back to binary integer values, the hexadecimal integer value corresponding to the array's thermal state, and the ASCII-encoded character the thermal state represents.

Fig. 7.7 shows a second signaling experiment. Here, the message `ATM` is transmitted, and the relative timing of the images is shown. For both experiments, an inter-character delay of 180 seconds (3 minutes) was chosen. The total elapsed time between the image marked "Start" and the image marked "End" was approximately 11 minutes. The relatively long symbol time was chosen due to the limitations of the TEC heat pumps, specifically the limited rate—discussed above—at which they can cool the antenna elements from a high temperature to a low temperature. Each antenna element takes roughly 50 seconds to slew from 20°C to 30°C, but roughly

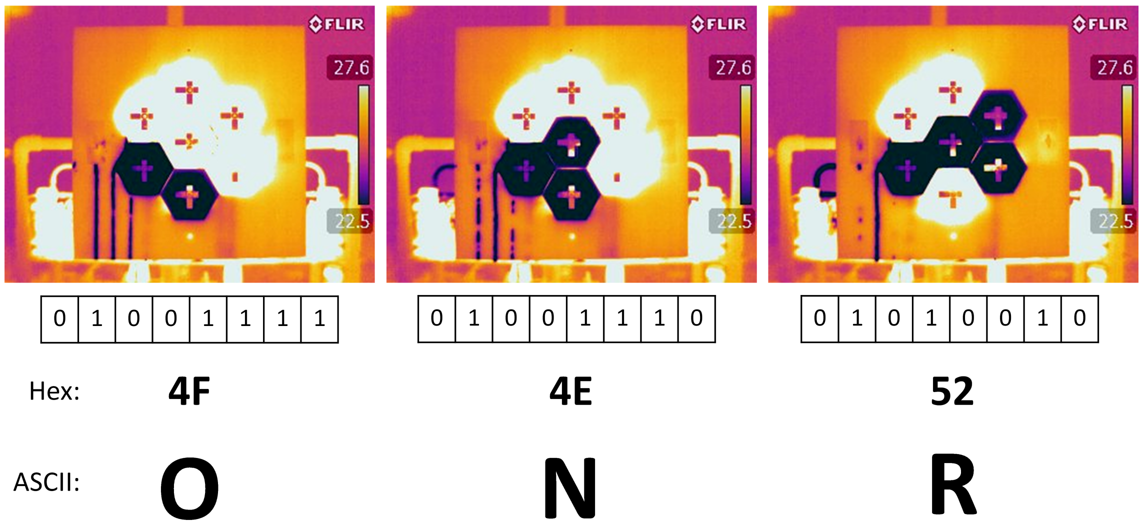


Fig. 7.6: ASCII character signaling using element temperatures

175 seconds to slew from 30°C to 20°C.

The relatively low temperature slew rates exhibited by the thermoregulation system can also be attributed to the high thermal capacity of the fluid heat exchange loop. Distilled water has a specific heat of 4.18 J g^{-1} K^{-1} and density of roughly 1 g mL^{-1} at 25°C. Thus, to change the temperature of the 60 mL of water in the heat exchange loop by 10°C requires the movement of roughly 2.5 kJ of heat energy. In the ideal case where the TEC provides $\dot{Q} = 40W$ and the heat exchange loop is otherwise adiabatic with respect to the ambient environment, the thermal loop would still take approximately 62 seconds to cool 10°C. A simple approach to reduce this time would be to replace the water with a liquid with lower specific heat. 3M's Fluorinert fluorocarbon oils would be an ideal substitute, as they are electrically insulating, highly thermally conductive, and have much lower specific heat capacities. (1.1 J g^{-1} K^{-1} for Fluorinert FC-3283) [59]

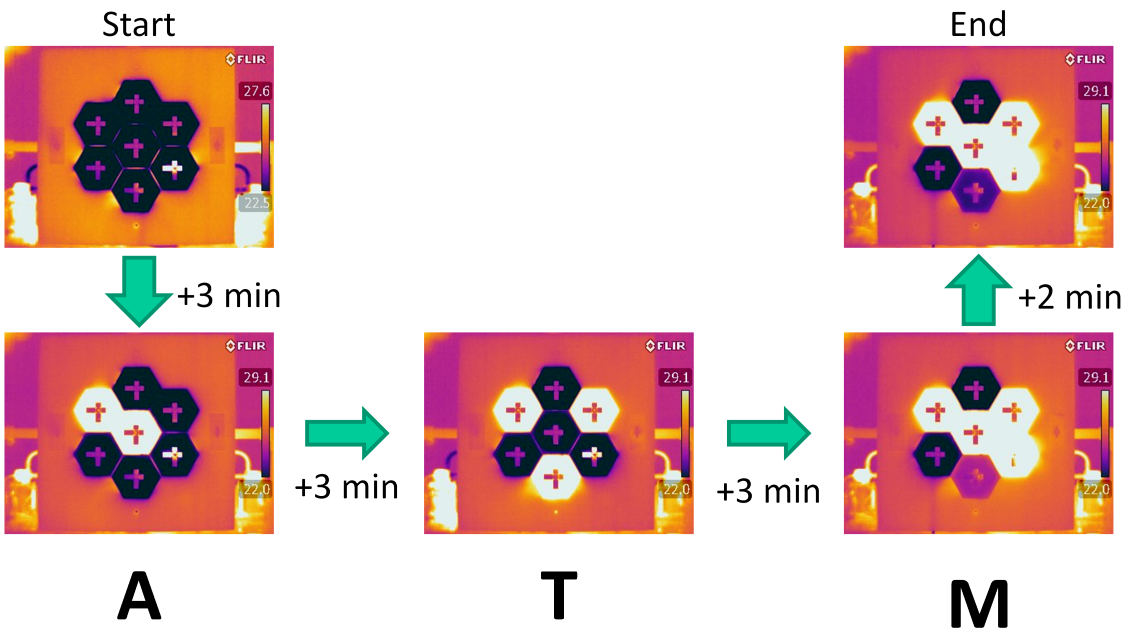
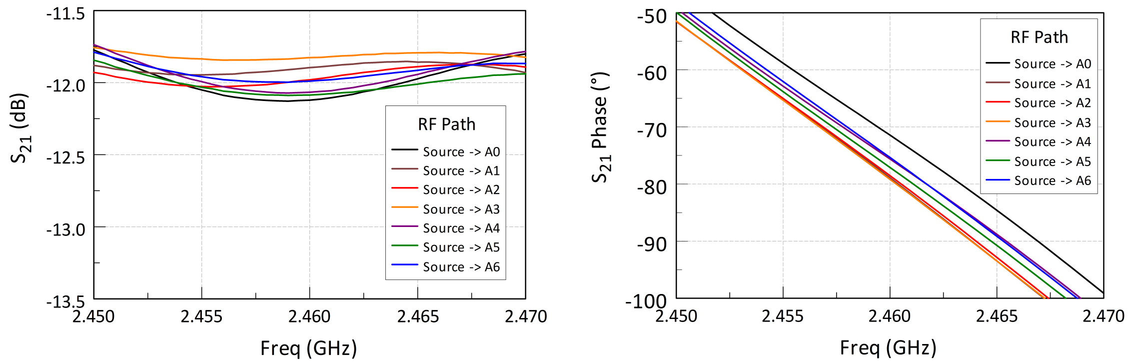


Fig. 7.7: Thermal signaling character timing

7.2 Phased Array Beamforming Experiment

The next set of experiments performed with the multifunctional EPRA array were a set of radiation pattern measurements to assess the array's ability to form and steer the main lobe of its radiation pattern (the beam). The array was excited with a uniform-amplitude corporate feed formed by a Mini-Circuits *ZB8PD-362-S+* 8-way broadband Wilkinson power divider, with port 8 terminated in a 50Ω load. First, the amplitude & phase balance of the feed network was profiled by measuring the forward transmission coefficient, S_{21}, from the common port on the power divider to each antenna element's SMA feed connector. These measurements are shown in Fig. 7.8. The corporate feed network is balanced to within 0.3dB and 10° phase across all seven RF paths.

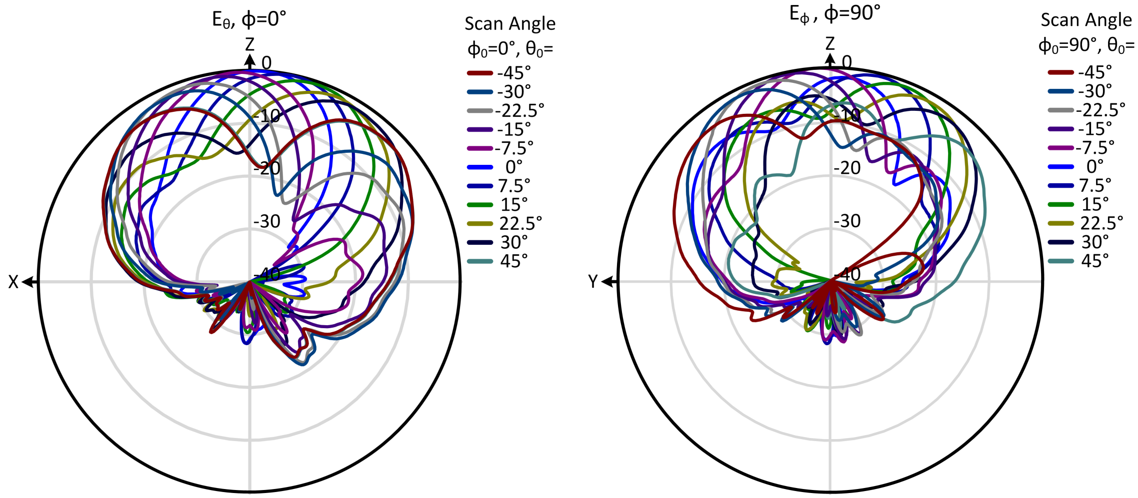


(a) Amplitude balance (b) Phase balance

Fig. 7.8: Amplitude & phase balance of phased array corporate feed network

Next, the array was commanded to steer the main lobe of its radiation pattern over a range of angles. Each beam angle is defined as an angle pair, (θ_0, ϕ_0) relative to the array's z-axis (normal to the plane of the array, as shown in Fig. 7.1). Thus, a steering angle of $(0°, 0°)$ corresponds to a broadside beam, and $(90°, 0°)$ corresponds to an end-fire beam in the X-direction. Excitation phase control is managed by the MCB of antenna module 0. The firmware accepts the beam steering command, computes the required phase shifts for each element in the array, and commands the DAC board to apply the correct bias voltage to each element's phase shifter.

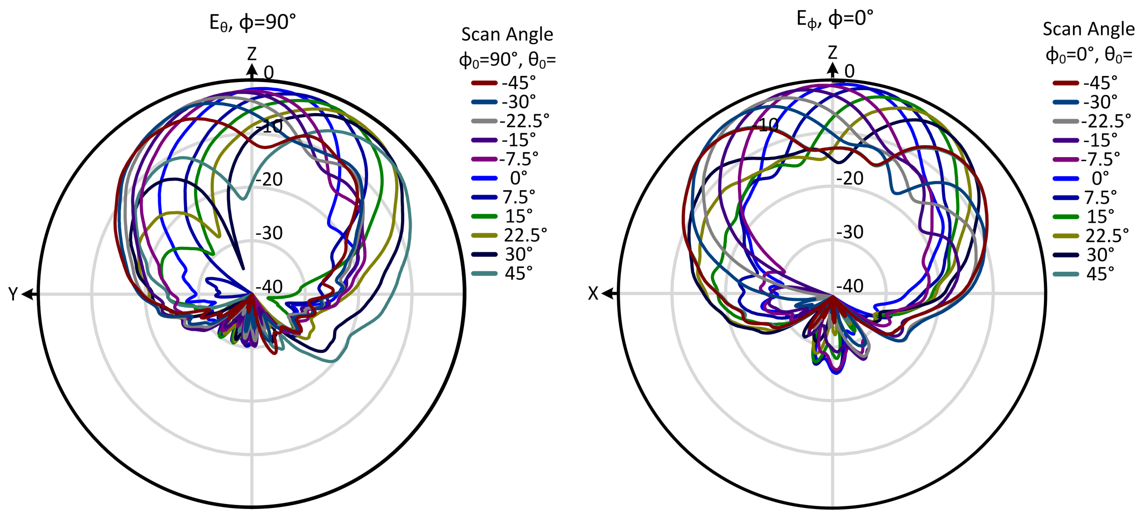
The first set of steering angles were chosen to demonstrate beam steering in the plane of the pattern measurement. The array is configured such that all elements are in either the X-polarized or Y-polarized state, and the steering angles are swept from -45° off broadside to +45° off broadside. Fig. 7.9 shows the in-plane steering results for the X-polarized mode. Fig. 7.10 shows the in-plane steering results for the Y-polarized mode. Excellent steering performance was observed from -30° to 30° in both polarization modes and in both primary measurement planes, with the measured main lobe peak steered to within 1.5° of the commanded angle. Beyond



(a) Beam steering on $\phi = 0°$ plane

(b) Beam steering on $\phi = 90°$ plane

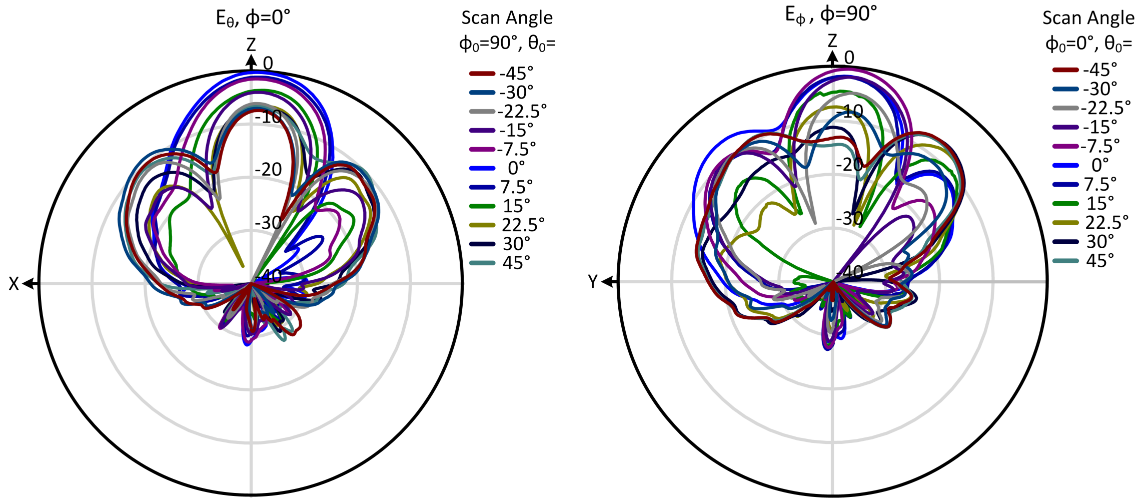
Fig. 7.9: Measured normalized array pattern: x-polarized mode, in-plane steering



(a) Beam steering on $\phi = 90°$ plane

(b) Beam steering on $\phi = 0°$ plane

Fig. 7.10: Measured normalized array pattern: y-polarized mode, in-plane steering

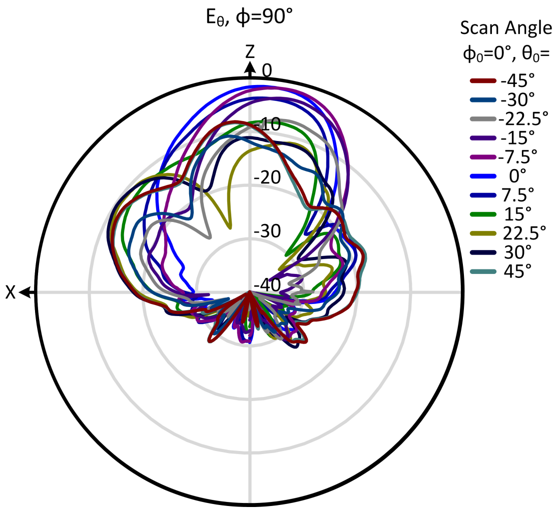


(a) Beam steering on $\phi = 90°$ plane (b) Beam steering on $\phi = 0°$ plane

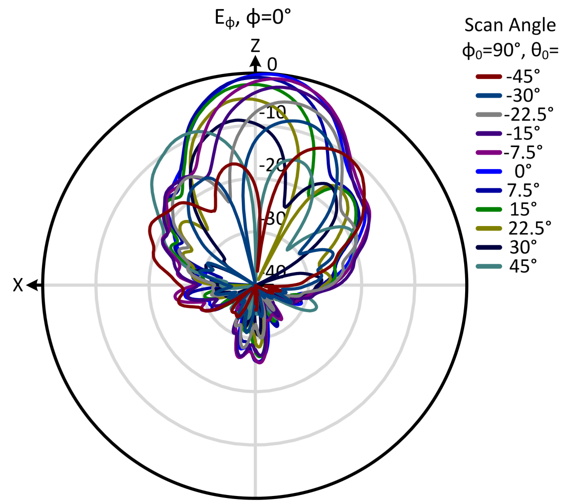
Fig. 7.11: Measured normalized array pattern: x-polarized mode, out-of-plane steering

±30°, the side lobe of the patterns grew to within 3 dB of the main lobe, and a difference of roughly 5° was observed between the measured and commanded beam angles. This behavior is to be expected, however, with an array of only 7 elements. Further, the high side lobes observed at beam angles off broadside are to be expected with an inter-element spacing of roughly 0.6λ.

Next, an equivalent set of pattern measurements were taken to evaluate the array's beamsteering performance when the beam is steered in a plane orthogonal to the measurement plane. Fig. 7.11 and 7.12 show the results of these *out-of-plane* steering tests. As expected, with larger steering angles off broadside the peak of the pattern stays at the same angle and the profile of the pattern stays roughly the same, with only the peak measured gain decreasing at larger positive & negative scan angles.

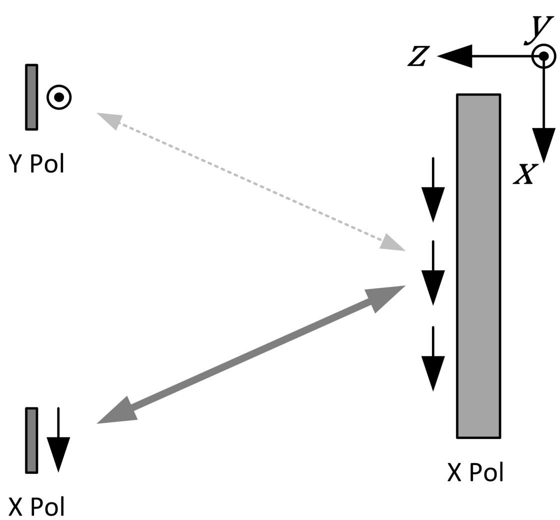


(a) Beam steering on $\phi = 0°$ plane

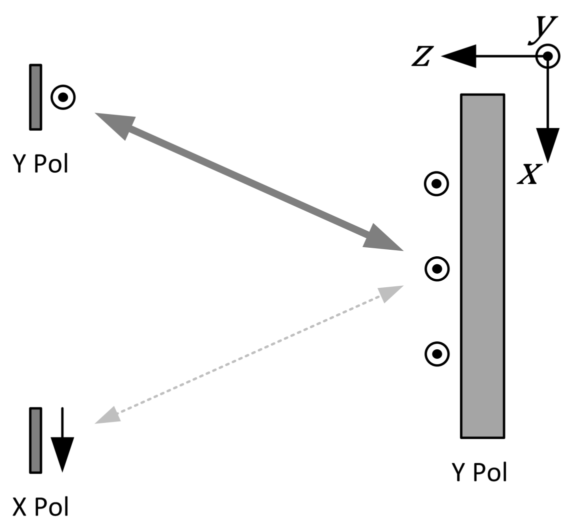


(b) Beam steering on $\phi = 90°$ plane

Fig. 7.12: Measured normalized array pattern: y-polarized mode, out-of-plane steering



(a) Array in x-polarization state



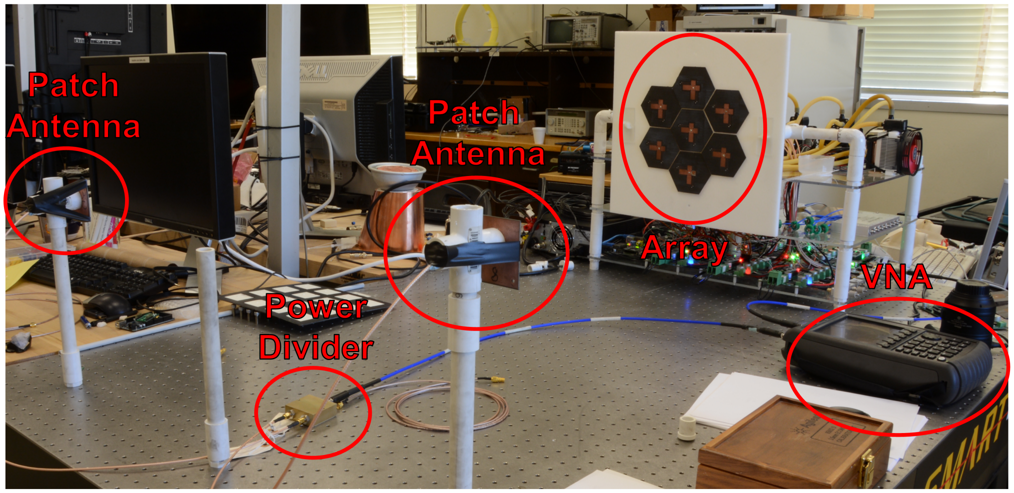
(b) Array in y-polarization state

Fig. 7.13: Conceptual overview of multiple emitter resolution using polarization reconfiguration

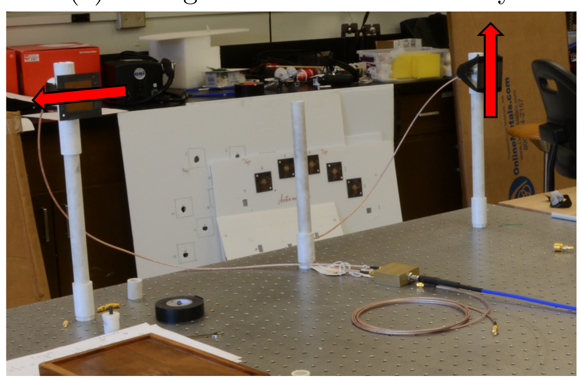
7.3 Multiple Emitter Direction-of-Arrival Estimation Experiment

The final set of experiments performed with the multifunctional EPRA array tested the polarization reconfigurability of the array in a direction-of-arrival estimation application. In particular, the goal of these experiments is to test whether the polarization reconfigurability of the EPRA array can resolve the individual arrival angles for two orthogonal, coherent signal emitters. Since the MUSIC algorithm requires *a priori* knowledge (or a guess) of the number of signal emitters to partition its search space into a signal subspace and a noise subspace, the resolution of multiple signal sources requires either knowledge of their number, or a method of filtering such that only a single emitter is considered at one time. Thus, it was hypothesized that the polarization reconfigurability of the array could provide this filtering to distinguish between orthogonal emitters. Furthermore, the mathematical basis of the MUSIC algorithm's subspace partitioning is based on the assumption that the only signal present besides that of the emitters is uncorrelated noise. Thus, attempting to resolve between two separate, coherent emitters by alternately partitioning one or the other into the noise subspace violates this assumption and exposes a weakness in the MUSIC algorithm. By implementing polarization switching to filter the orthogonal emitters, it is hypothesized that the MUSIC algorithm will be able to distinguish them. A pictorial representation of this concept is shown in Fig. 7.13, where the thick gray arrows represent the stronger link path between the array and co-polarized emitter.

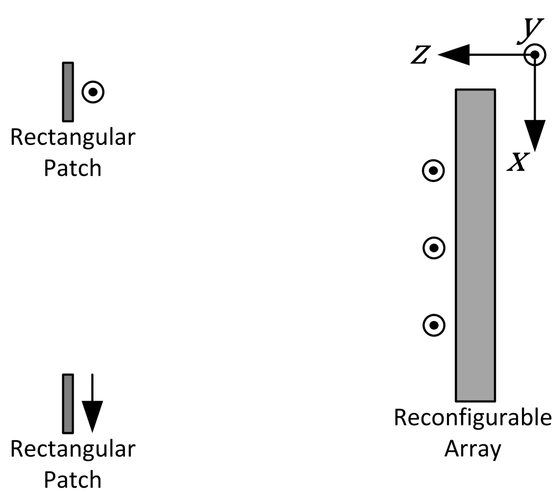
The experimental configuration used in this series of tests is shown in Fig. 7.14. Real-time direction-of-arrival estimation of a moving emitter with an antenna array requires a set of synchronized, phase-sensitive receivers for each antenna element in order to record the relative phase difference of the received signal at each an-



(a) Facing toward the EPRA array

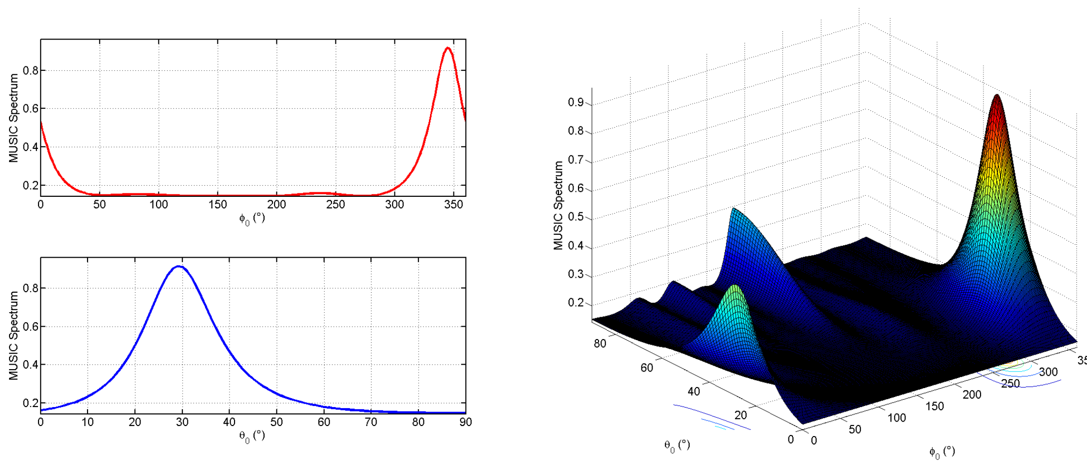


(b) Facing away from the EPRA array



(c) Overhead schematic view

Fig. 7.14: Polarization-reconfigurable direction-of-arrival estimation experimental setup



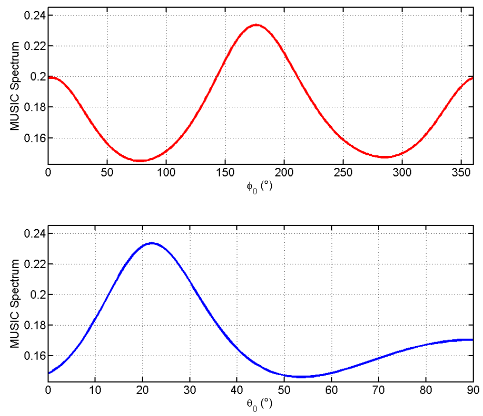
(a) Cutplanes through spectrum peak

(b) Full 2D MUSIC spectrum

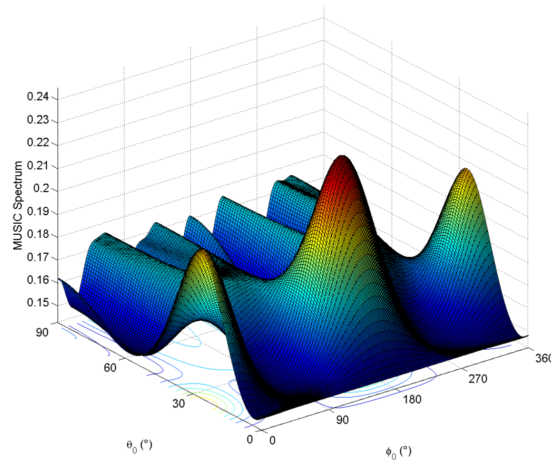
Fig. 7.15: MUSIC pseudospectra: x-polarized array

tenna element. Since a set of 7 such receivers were not available for use when these experiments were performed, the experiment was performed as a quasi-static approximation. An Agilent Technologies N9923A *FieldFox* portable vector network analyzer (VNA) was used as a substitute for the emitter source and phase-sensitive receiver. The VNA was configured to measure the forward transmission coefficient, S_{21}. The source port (port 1) was connected to a 2-way Wilkinson power divider, the outputs of which fed the two stationary, orthogonal emitter antennas through phase-matched coaxial cables, shown in Fig. 7.14b. The receive port (port 2) of the VNA was then connected sequentially to the feed of each antenna element in the EPRA array, and a set of 16 measurement sweeps were averaged and recorded for each antenna in the array. The phases of the complex forward transmission coefficient between the emitters and each element of the array were then used as the angle argument for a set of equal-amplitude signal vectors which formed the input to the MUSIC algorithm.

The results of the direction-of-arrival estimation experiments are shown in Figs.



(a) Cutplanes through spectrum peak



(b) Full 2D MUSIC spectrum

Fig. 7.16: MUSIC pseudospectra: y-polarized array

7.15 and 7.16. In both cases, the MUSIC algorithm was set to search only in the $+z$ hemisphere (i.e. only in front of the array) with a $1°$ search increment in θ and ϕ. Further, the algorithm is set to only partition one signal into the signal subspace. In Fig. 7.15, the array is X-polarized and the experimental conditions match those displayed in Fig. 7.13a. The peak in the MUSIC spectrum occurs at $(\theta_0, \phi_0) = (29°, 345°)$. In Fig. 7.16, the experimental setup is the same, except the EPRA elements in the array are all in their Y-polarized state, as in Fig. 7.13b. The peak in the MUSIC spectrum for the Y-polarized case occurs at $(\theta_0, \phi_0) = (22°, 176°)$. In both cases, the calculated pseudospectra show an unambiguous peak at a single direction in the search space. In the Y-polarized measurement, a relatively large secondary peak appears at $(\theta_0, \phi_0) = (22°, 2°)$. This is likely attributable to the signal from the X-polarized antenna coupling into the array. The experimental data was collected with both the array and emitter antennas over a conductive steel optical table, which likely generated significant multipath propagation effects. Nevertheless, the estimated direction-of-arrival for both emitter antennas agrees well with the

Table 7.2: Comparison of MUSIC-estimated and physically measured direction of arrival

	MUSIC DoA Estimation		**Physical Measurement**	
	θ_0	ϕ_0	θ_0	ϕ_0
X-Polarized Emitter	29°	345°	23°	0/360°
Y-Polarized Emitter	22°	176°	23°	180°

physically measured experimental setup. Table 7.2 shows a summary of the actual and estimated DoA for both emitters.

8. SUMMARY

8.1 Conclusion

In conclusion, this work has demonstrated the successful conception, implementation, and evaluation of a modular, multifunctional reconfigurable antenna array. A system-level design was successfully implemented incorporating a modular control system, novel multifunctional antenna element design, and networked control system. The control architecture demonstrated is dynamically scalable and extensible, and provides real-time, closed-loop control of the reconfiguration mechanisms of the array. The array testbed demonstrates the type & number of reconfiguration control mechanisms necessary to utilize a full array of fluidic-reconfigurable antenna elements.

The modular, multifunctional array testbed was successfully tested in a variety of wireless applications: polarization reconfiguration, beamforming & beamsteering, and direction-of-arrival estimation were all demonstrated. Furthermore, a novel themal-IR raster signaling system was conceptualized and implemented with a proof-of-concept demonstration. This technique shows promise to facilitate the multifunctionalization of large planar antenna arrays.

8.2 Future Work

This thesis presents a number of different opportunities to pursue future work. First, the material systems and material handling strategies for functionalizing the fluidic-reconfigurable tri-band polarization- & frequency reconfigurable antenna design present a range of future research opportunities. There is considerable room to optimize the electromagnetic design of the TBPFRA element, as well.

The thermal-IR signaling system also presents a number of opportunities for continued research & development. Alternative heat pumping mechanisms could be explored to achieve a faster signaling rate. More advanced data encoding & modulation schemes also present opportunities to improve upon the proof-of-concept demonstration presented here.

REFERENCES

[1] C. A. Balanis, *Antenna Theory: Analysis and Design*, 3rd ed. Hoboken, NJ: John Wiley & Sons, 2005.

[2] D. Schaubert, F. Farrar, A. Sindoris, and S. Hayes, "Microstrip antennas with frequency agility and polarization diversity," *IEEE Trans. Antennas Propag.*, vol. 29, no. 1, pp. 118–123, Jan 1981. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1142546

[3] P. Bhartia and I. Bahl, "A frequency agile microstrip antenna," in *Antennas and Propagation Society International Symposium*, vol. 20, May 1982, pp. 304–307. [Online]. Available: http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=1148900

[4] Y. Lo and S. W. Lee, *Antenna Handbook: Theory, Applications, and Design.* New York, NY: Van Nostrand Reinhold, 1988.

[5] R. Munson, "Conformal microstrip antennas and microstrip phased arrays," *IEEE Trans. Antennas Propag.*, vol. 22, no. 1, pp. 74–78, Jan 1974. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1140723

[6] C. A. Balanis, *Advanced Engineering Electromagnetics*, 2nd ed. Hoboken, NJ: John Wiley & Sons, 2012.

[7] H. Wheeler, "Transmission-line properties of a strip on a dielectric sheet on a plane," *IEEE Trans. Microw. Theory Tech.*, vol. 25, no. 8, pp. 631–647, 1977. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1129179

[8] E. Hammerstad, “Equations for microstrip circuit design,” in *5th European Microwave Conference*, 1975, pp. 268–272. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4130821

[9] B. Li and Q. Xue, “Polarization-reconfigurable omnidirectional antenna combining dipole and loop radiators,” *IEEE Antennas Wireless Propag. Lett.*, vol. 12, pp. 1102–1105, 2013. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6600984

[10] M. Parihar, A. Basu, and S. Koul, “Polarization reconfigurable microstrip antenna,” in *Asia Pacific Microwave Conference*, 2009, pp. 1918–1921. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5384516

[11] P.-Y. Qin, Y. Guo, and C. Ding, “A microstrip dual-band polarization reconfigurable antenna,” in *IEEE Antennas and Propagation Society International Symposium (APSURSI)*, 2013, pp. 1640–1641. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6711479

[12] X.-X. Yang, B.-C. Shao, F. Yang, A. Elsherbeni, and B. Gong, “A polarization reconfigurable patch antenna with loop slots on the ground plane,” *IEEE Antennas Wireless Propag. Lett.*, vol. 11, pp. 69–72, 2012. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6121937

[13] R.-H. Chen and J.-S. Row, “Single-fed microstrip patch antenna with switchable polarization,” *IEEE Trans. Antennas Propag.*, vol. 56, no. 4, pp. 922–926, 2008. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=4483613

[14] J.-F. Tsai and J.-S. Row, “Reconfigurable square-ring microstrip antenna,” *IEEE Trans. Antennas Propag.*, vol. 61, no. 5, pp. 2857–2860, 2013. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6425417

[15] S. Hum and H. Y. Xiong, "Analysis and design of a differentially-fed frequency agile microstrip patch antenna," *IEEE Trans. Antennas Propag.*, vol. 58, no. 10, pp. 3122–3130, 2010. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5498967

[16] M. Kehn, O. Quevedo-Teruel, and E. Rajo-Iglesias, "Reconfigurable loaded planar inverted-f antenna using varactor diodes," *IEEE Antennas Wireless Propag. Lett.*, vol. 10, pp. 466–468, 2011. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5765653

[17] Y. Yu, J. Xiong, H. Li, and S. He, "An electrically small frequency reconfigurable antenna with a wide tuning range," *IEEE Antennas Wireless Propag. Lett.*, vol. 10, pp. 103–106, 2011. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5712155

[18] T. Jung, I.-J. Hyun, J.-M. Kim, C.-W. Baek, and S. Lim, "Polarization reconfigurable antenna on RF-MEMS packaging platform," in *Asia Pacific Microwave Conference*, 2009, pp. 579–582. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5384160

[19] T. J. Jung, I.-J. Hyeon, C.-W. Baek, and S. Lim, "Circular/linear polarization reconfigurable antenna on simplified RF-MEMS packaging platform in k-band," *IEEE Trans. Antennas Propag.*, vol. 60, no. 11, pp. 5039–5045, 2012. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6237506

[20] G. Huff and J. Bernhard, "Integration of packaged RF MEMS switches with radiation pattern reconfigurable square spiral microstrip antennas," *IEEE Trans. Antennas Propag.*, vol. 54, no. 2, pp. 464–469, 2006. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1589416

[21] C.-Y. Chiu, J. Li, S. Song, and R. Murch, "Frequency-reconfigurable pixel slot antenna," *IEEE Trans. Antennas Propag.*, vol. 60, no. 10, pp. 4921–4924, 2012. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6236033

[22] C. Murray and R. Franklin, "Frequency tunable fluidic annular slot antenna," in *IEEE Antennas and Propagation Society International Symposium (APSURSI)*, 2013, pp. 386–387. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6710854

[23] G. Huff and S. Goldberger, "A coaxial stub microfluidic impedance transformer (COSMIX)," *IEEE Microw. Wireless Compon. Lett.*, vol. 20, no. 3, pp. 154–156, 2010. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5406008

[24] S. Long and G. Huff, "A fluidic loading mechanism for phase reconfigurable reflectarray elements," *IEEE Antennas Wireless Propag. Lett.*, vol. 10, pp. 876–879, 2011. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5996072

[25] ——, "Experiments on a fluidic loading mechanism for beam-steering reflectarrays," in *IEEE International Conference on Wireless Information Technology and Systems (ICWITS)*, 2010, pp. 1–4. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5611852

[26] G. Huff, D. Rolando, P. Walters, and J. McDonald, "A frequency reconfigurable dielectric resonator antenna using colloidal dispersions," *IEEE Antennas Wireless Propag. Lett.*, vol. 9, pp. 288–290, 2010. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=5438799

[27] A. King, J. Patrick, N. Sottos, S. White, G. Huff, and J. Bernhard, "Microfluidically switched frequency-reconfigurable slot antennas," *IEEE Antennas Wireless Propag. Lett.*, vol. 12, pp. 828–831, 2013. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6547156

[28] M. Kelley, C. Koo, H. Mcquilken, B. Lawrence, S. Li, A. Han, and G. Huff, "Frequency reconfigurable patch antenna using liquid metal as switching mechanism," *IET Electronics Letters*, vol. 49, no. 22, pp. 1370–1371, 2013. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6649516

[29] C. Kitamura, A. Morishita, T. Chun, W. Tonaki, A. Ohta, and W. Shiroma, "A liquid-metal reconfigurable Yagi-Uda monopole array," in *IEEE MTT-S International Microwave Symposium Digest (IMS)*, 2013, pp. 1–3. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6697779

[30] A. Morishita, C. Kitamura, A. Ohta, and W. Shiroma, "A liquid-metal monopole array with tunable frequency, gain, and beam steering," *IEEE Antennas Wireless Propag. Lett.*, vol. 12, pp. 1388–1391, 2013. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6645428

[31] G. Hayes, J.-H. So, A. Qusba, M. Dickey, and G. Lazzi, "Flexible liquid metal alloy (EGaIn) microstrip patch antenna," *IEEE Trans. Antennas Propag.*, vol. 60, no. 5, pp. 2151–2156, 2012. [Online]. Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6182698

[32] J. Barrera and G. Huff, "A fluidic loading mechanism in a polarization reconfigurable antenna with a comparison to solid state approaches," *IEEE Trans. Antennas Propag.*, vol. PP, no. 99, pp. 1–1, 2014. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6819830

[33] S. Goldberger, F. Drummond, J. Barrera, S. Davis, J. Edelen, M. Geppert, Y. Judie, Q. Manley, C. Peters, S. Smith *et al.*, "Switchable antenna polarization using surface-integrated fluidic loading mechanisms," in *24th Annual AIAA/USU Conference on Small Satellites.* Logan, UT: American Institute of Aeronautics and Astronautics & Utah State University, 2010. [Online]. Available: http://digitalcommons.usu.edu/smallsat/2010/all2010/62/

[34] R. Clarke. Dielectric properties of materials. Online. National Physical Laboratory. Middlesex, UK. [Online]. Available: http://www.kayelaby.npl.co.uk/general_physics/2_6/2_6_5.html

[35] *Fluorinert (TM) Electronic Liquid FC-70*, 3M Performance Materials, St. Paul, MN, 2000. [Online]. Available: http://multimedia.3m.com/mws/mediawebserver?mwsId=66666UgxGCuNyXTtnxTE5Xz6EVtQEcuZgVs6EVs6E666666--&fn=prodinfo_FC70.pdf

[36] V. Meriakri, E. Chigrai, and M. Parkhomenko, *The science and technology of millimetre wave components and devices*, ser. Electrocomponent science monograph. London, UK: Taylor & Francis, 2002, vol. 12, p. 163.

[37] M. Larhed, C. Moberg, and A. Hallberg, "Microwave-accelerated homogeneous catalysis in organic chemistry," *Accounts of Chemical Research*, vol. 35, no. 9, pp. 717–727, 2002. [Online]. Available: http://pubs.acs.org/doi/abs/10.1021/ar010074v

[38] A. Gregory, R. Clarke, and M. Cox, "Traceable measurement of dielectric reference liquids over the temperature interval 10–50 C using coaxial-line methods," *Meas. Sci. Technol.*, vol. 20, no. 7, 2009. [Online]. Available: http://iopscience.iop.org/0957-0233/20/7/075106

[39] F. Gross, *Smart antennas for wireless communications: with MATLAB*. New York, NY: McGraw-Hill, 2005.

[40] A. Vesa, "Direction of arrival estimation using MUSIC and Root–MUSIC algorithm," in *18th Telecommunication Forum, TELFOR*. Belgrade, Serbia: Telecommunications Society, Belgrade, 2010.

[41] *CP Series Application Notes*, CUI, Inc, Tualatin, Oregon, 2009. [Online]. Available: http://www.cui.com/product/resource/peltier-application-notes.pdf

[42] N. Minorsky, "Directional stability of automatically steered bodies," *Journal of ASNE*, vol. 42, no. 2, pp. 280–309, 1922.

[43] K. J. Åström. (2002) Lecture notes for ME 155A, Control System Design. Department of Mechanical and Environmental Engineering, University of California Santa Barbara. Santa Barbara, CA. [Online]. Available: http://www.cds.caltech.edu/~murray/courses/cds101/fa02/caltech/astrom.html

[44] *AVR221: Discrete PID controller*, Atmel, Inc, San Jose, CA, 2006. [Online]. Available: http://www.atmel.com/images/doc2558.pdf

[45] A. Visioli, B. M. Ayyub, G. J. Klir, S. Dineen, D. Cox, S. Kotz, J. van Dorp, A. Barbour, L. Chen, E. J. Billo *et al.*, "Practical PID control," 2008. [Online]. Available: http://www.tandfonline.com/doi/pdf/10.1198/004017008000000172

[46] A. Visioli, "Modified anti-windup scheme for PID controllers," *IEE Proceedings-Control Theory and Applications*, vol. 150, no. 1, pp. 49–54, 2003. [Online]. Available: http://digital-library.theiet.org/content/journals/10.1049/ip-cta_20020769

[47] M. Shahrokhi and A. Zomorrodi, "Comparison of PID controller tuning methods," in *Department of Chemical & Petroleum Engineering, Sharif University*

of Technology, Tehran, Iran, 2013.

[48] *IEEE Standard Letter Designations for Radar-Frequency Bands*, IEEE Std. IEEE Std 521-2002 (Revision of IEEE Std 521-1984), 2003. [Online]. Available: http://ieeexplore.ieee.org/servlet/opac?punumber=8332

[49] 450° *Analog Phase Shifter, 2-4 GHz*, Hittite Microwave Corporation, Chelmsford, MA, 2011. [Online]. Available: https://www.hittite.com/content/documents/data_sheet/hmc928lp5.pdf

[50] T. Hutter, W.-A. C. Bauer, S. R. Elliott, and W. T. Huck, "Formation of spherical and non-spherical eutectic gallium-indium liquid-metal microdroplets in microfluidic channels at room temperature," *Advanced Functional Materials*, vol. 22, no. 12, pp. 2624–2631, 2012. [Online]. Available: http://onlinelibrary.wiley.com/doi/10.1002/adfm.201200324/full

[51] P. Cannon, L. St. Pierre, and A. Miller, "Solubilities of hydrogen and oxygen in polydimethylsiloxanes." *Journal of Chemical and Engineering Data*, vol. 5, no. 2, pp. 236–236, 1960. [Online]. Available: http://pubs.acs.org/doi/abs/10.1021/je60006a027

[52] D. M. Pozar, *Microwave engineering*, 4th ed. Hoboken, NJ: John Wiley & Sons, 2012.

[53] *8-bit Atmel Microcontroller with 64/128Kbytes of ISP Flash and USB Controller AT90USB646 AT90USB647 AT90USB1286 AT90USB1287*, Atmel, Inc, San Jose, CA, 2012. [Online]. Available: http://www.atmel.com/Images/doc7593.pdf

[54] *2.5V and 4.096V Voltage References*, Microchip Technology, Chandler, AZ, 2001. [Online]. Available: http://ww1.microchip.com/downloads/en/

DeviceDoc/21653a.pdf

[55] J. S. Steinhart and S. R. Hart, "Calibration curves for thermistors," in *Deep Sea Research and Oceanographic Abstracts*, vol. 15, no. 4. Elsevier, 1968, pp. 497–503. [Online]. Available: http://www.sciencedirect.com/science/article/pii/0011747168900570

[56] J. Jensen, "A cognitive phased array using smart phone control," Master's thesis, Texas A&M University, College Station, TX, May 2012. [Online]. Available: http://hdl.handle.net/1969.1/ETD-TAMU-2012-05-10856

[57] P. J. Stoffregen. (2014) Teensy USB development board. PJRC. [Online]. Available: http://www.pjrc.com/teensy/index.html

[58] Advanced Circuits, Aurora, CO.

[59] *Fluorinert (TM) Electronic Liquid FC-3283*, 3M Performance Materials, St. Paul, MN, 2001. [Online]. Available: http://multimedia.3m.com/mws/mediawebserver?mwsId=66666UgxGCuNyXTtnxTEoxF6EVtQEcuZgVs6EVs6E666666--&fn=prodinfo_FC3283.pdf

APPENDIX A
MCB FIRMWARE SOURCE CODE

The following is a reproduction of the C source code which forms the firmware for the *Teensy 2.0++* MCU on the MCB. The source files are grouped into sections by the logical nature of the functions they contain. These sections are as follows:

- **Main Code**: The source files containing the main loop of the firmware, startup procedures, real-time temperature control algorithm, and interrupt routines
- **Command Processing**: The source files containing the library which parses incoming text commands, and the functions called by that library to handle execution of those commands
- **Peripheral Control Code**: The source files containing the low- and mid-level drivers for the on-chip and off-chip peripherals, including the ADC, PWM generators, and off-chip DAC
- **Communication Code**: The source files containing low- and mid-level drivers for the communication peripherals, including the XBee WiFi radio, UART, USB serial endpoint, and SPI transceiver

A.1 Main Code

A.1.1 Powerup Initialization, Closed-Loop Temperature Control, & Main Loop Functions

AntennaController_v2_1.h

```c
/*
Global variables & data structures for Antenna Controller
 */


#ifndef ANTENNACONTROLLER_V2_1_H_
#define ANTENNACONTROLLER_V2_1_H_

#include <stdint.h>

// Device serial number string for XBee initialization
extern char deviceSerial[];

// Temperature sense source mode
enum temp_sense_src {TEC, MODULE};
extern enum temp_sense_src myTempSenseSource;

// Global flag bits for various control functions
struct flag_bits {
   uint8_t  tmr0_ovf :1;
   uint8_t  ext_tgt     :1;
   uint8_t  in_scram :1;
   uint8_t  auto_mode   :1;
   uint8_t  bit4     :1;
   uint8_t  bit5     :1;
   uint8_t  bit6     :1;
   uint8_t debug     :1;
   };
extern volatile struct flag_bits globalFlags;

// PID gains & integrator max for PID initialization
/*
   For CUI CP85438 TEC w/ Rosewill RCX-Z80-AL heatsink/fan
   and XSPC XBOX360Slim waterblock only, experimentally determined
   parameters are:

   k_p = 2442
   k_i = 68
   k_d = 13255

   With 250ms sample time

   k_p = K_c
   k_i = K_c * T_s/T_i
   k_d = K_c * T_d/T_s

   K_c is proportional gain
   T_i is integration time
   T_d is derivative time
   T_s is sample interval time
*/
struct pid_params {
   int_fast16_t k_p;
   int_fast16_t k_i;
   int_fast16_t k_d;
   int_fast32_t Imax;
   };
```

```
extern struct pid_params myPIDparams;

// Struct of pulse widths for each valve's 3 positions
struct valve_params {
   uint8_t pos0;
   uint8_t pos1;
   uint8_t pos2;
   };

extern struct valve_params myValveParams[8];

// Struct of speed and delay for each pump output
struct pump_params {
   uint_fast16_t pw;
   uint_fast16_t timer_ms;
   };

extern struct pump_params myPumpParams[4];

// PID runtime data storage structure as defined in pid.h
extern struct pid_data myPIDdata;

// Measured temperature & target temperature variables
extern volatile int_fast16_t t_tec;
extern volatile int_fast16_t t_mod;
extern volatile int_fast16_t t_ref;
extern volatile int_fast16_t t_sink;
extern int_fast16_t t_tgt;


//Uncomment to switch communication from UART to USB serial port for debugging:
//#define USB_SERIAL

// Various control settings:
#define MAX_TEMP        670      // Maximum safe temperature (in 1/10C)
#define MIN_TEMP        -50      // Minimum temperature (in 1/10C)
#define MAX_SET_TEMP    500      // Maximum allowed temperature setting (in 1/10C)
#define MIN_SET_TEMP    -50      // Minimum allowed temperature setting (in 1/10C)
#define baudrate        9600     // UART baud rate
#define ADC_AVGS        5        // Number of ADC readings to take when measuring temps
#define manageTemp_cnt_ovf 250   // Temperature measurement/management interval in ms
#define SERVO_TIME         750   // Servo actuation time delay in ms

// EEPROM addresses for custom per-module parameters:

// struct myPIDparams > PID parameters in myPIDparams
#define EEP_ADDRESS_PID    (void*)0x10

// struct myValveParams[8] > Servo valve position:pulse width map
#define EEP_ADDRESS_VALVE  (void*)(EEP_ADDRESS_PID + sizeof(myPIDparams))

// struct myPumpParams[4] > Pump parameters
#define EEP_ADDRESS_PUMP   (void*)(EEP_ADDRESS_VALVE + sizeof(myValveParams))

#endif /* ANTENNACONTROLLER_V2_1_H_ */
```

AntennaController_v2_1.c

```
/*
Antenna Controller

Communication & control via XBee WiFi S6 running firmware v.102D
PID thermal control for Peltier TEC with selectable temperature feedback
8-way hobby servo valve control
```

```
4-way pump control

 */

#include <stdint.h>
#include <stdio.h>
#include <math.h>
#include <avr/io.h>
#include <avr/pgmspace.h>
#include <util/delay.h>
#include <avr/interrupt.h>
#include <avr/eeprom.h>
#include <stdlib.h>

#include "AntennaController_v2_1.h"
#include "adc2temp.h"
#include "pwm.h"
#include "parse_cmd.h"
#include "adc.h"
#include "xbee.h"
#include "uart.h"
#include "serial.h"
#include "pid.h"
#include "spi.h"
#include "AD5668.h"

#define LED_CONFIG      (DDRD |= (1<<6))
#define LED_ON       (PORTD |= (1<<6))
#define LED_OFF         (PORTD &= ~(1<<6))
#define CPU_PRESCALE(n) (CLKPR = 0x80, CLKPR = (n))

// ADC Pin Names
#define tec_therm PF0
#define modu_therm   PF1
#define ref_therm PF2
#define sink_therm   PF3
#define ADC4      PF4

char deviceSerial[17];     // Character array for XBee serial number

volatile uint8_t manageTemp_cnt; //

enum temp_sense_src myTempSenseSource = TEC;

volatile struct flag_bits globalFlags;


int_fast16_t t_tgt_init = 250;   // Initial temperature setpoint = 25.0C

struct pid_data myPIDdata;

struct pid_params myPIDparams;
struct valve_params myValveParams[8];
struct pump_params myPumpParams[4];

volatile int_fast16_t t_tec;
volatile int_fast16_t t_mod;
volatile int_fast16_t t_ref;
volatile int_fast16_t t_sink;
int_fast16_t t_tgt;

volatile int_fast16_t t_sys = 0;
volatile int_fast16_t t_set = 0;

```

```
// Reads parameter structures out of EEPROM into SRAM variables
// Called on startup
/************************************************************************/
/* NOTE: EEPROM values on uninitialized MCUs must be programmed         */
/*       using SETPID WRITE and SETVALVEPARAM WRITE commands            */
/************************************************************************/
void init_parameters() {
   eeprom_read_block(&myPIDparams, EEP_ADDRESS_PID, sizeof(myPIDparams));
   eeprom_read_block(&myValveParams, EEP_ADDRESS_VALVE, sizeof(myValveParams));
   eeprom_read_block(&myPumpParams, EEP_ADDRESS_PUMP, sizeof(myPumpParams));
   for (int i = 1; i < 5; i++) {
      set_pump(i, MODE_OFF, myPumpParams[i-1].pw);
   }
}

// TEC temperature control function
// Reads thermistor ADCs, updates global temperature variables
// calls PID algorithm, & updates TEC output H-bridge PWM
void manageTemp() {
   LED_ON;
   uint_fast16_t t_tec_read = 0;
   uint_fast16_t t_mod_read = 0;
   uint_fast16_t t_ref_read = 0;
   uint_fast16_t t_sink_read = 0;

   for (int i = 0; i < ADC_AVGS; i++) {
      t_tec_read += read_adc(tec_therm);
      t_mod_read += read_adc(modu_therm);
      t_ref_read += read_adc(ref_therm);
      t_sink_read += read_adc(sink_therm);
   }

   t_tec = adc2temp(t_tec_read/ADC_AVGS);
   t_mod = adc2temp(t_mod_read/ADC_AVGS);
   t_ref = adc2temp(t_ref_read/ADC_AVGS);
   t_sink = adc2temp(t_sink_read/ADC_AVGS);

   // Check if temperature limit exceeded & SCRAM if so
   if ((t_tec > MAX_TEMP) || (t_sink > MAX_TEMP) || (t_mod > MAX_TEMP)) {
      scram();
   }
   if ((t_tec < MIN_TEMP) || (t_sink < MIN_TEMP) || (t_mod < MIN_TEMP)) {
      scram();
   }

   // Only run PID control if we're in auto and not in SCRAM
   if ((!(globalFlags.in_scram) && globalFlags.auto_mode)) {
      //int_fast16_t t_sys, t_set;
      // Choose system temperature based on selected source
      switch (myTempSenseSource) {
         case TEC:
            t_sys = t_tec;
            break;
         case MODULE:
            t_sys = t_mod;
            break;
         default:
            t_sys = 0;
            break;
      }
      // Choose setpoint temperature based on selected source
      switch (globalFlags.ext_tgt) {
         case 0:
```

```
            t_set = t_tgt;
            break;
         case 1:
            t_set = t_ref;
            break;
         default:
            t_set = 0;
            break;
      }

      // Generate PID control output
      int_fast16_t pid_output = pid_control(t_set, t_sys, &myPIDdata);
      uint_fast16_t pw = (abs(pid_output)>>5);  // Max pulse width is 1023

      if    (pid_output > 0)  set_tec(TEC_HEAT, pw);
      else if (pid_output == 0)  set_tec(TEC_OFF, 0);
      else if (pid_output < 0)   set_tec(TEC_COOL, pw);
   }

   // If we're SCRAMmed or not in auto, reset the PID integrator
   else if ((globalFlags.in_scram || !(globalFlags.auto_mode))) {
      pid_reset_integrator(&myPIDdata);
   }

   LED_OFF;
}

void setup(void) {
   cli();   // Global Interrupt Disable

   LED_CONFIG;
   CPU_PRESCALE(0);

#ifdef USB_SERIAL
   usb_init();
   while (!usb_configured()) ; // Wait for USB serial port ready
#else
   uart_init(baudrate);
#endif

   _delay_ms(200);

   LED_ON;

   configure_pwm();
   configure_adc();

   configure_spi();
   //_delay_us(100);
   //configure_dac();   // Must be called after configure_spi()

   init_parameters();

   pid_init(myPIDparams.k_p, myPIDparams.k_i, myPIDparams.k_d, &myPIDdata);

   #ifndef USB_SERIAL
   configure_xbee();
   #endif

   setup_SerialCommand();

   t_tgt = t_tgt_init;
```

```c
    sei();   // Global Interrupt Enable
}

// Timer 0 Overflow Interrupt Routine
// Runs at 1kHz
ISR(TIMER0_OVF_vect) {
    if (manageTemp_cnt < manageTemp_cnt_ovf - 1) {manageTemp_cnt++;}
    else {
        manageTemp_cnt = 0;
        globalFlags.tmr0_ovf = 1;
    }
}

ISR(INT0_vect) {

}

ISR(INT1_vect) {

}

ISR(INT6_vect) {

}

ISR(INT7_vect) {

}

int main(void) {
    setup();
    while (1) {
        serialCommand_readSerial();
        if (globalFlags.tmr0_ovf) {
            manageTemp();
            globalFlags.tmr0_ovf = 0; //Reset TIMER0 overflow flag
        }
    }
}
```

A.1.2 Discrete PID Control Algorithm

pid.h

```c
/*
 * pid.h
 *
 * Created: 11/24/2013 6:43:38 PM
 *  Author: Nick
 */


#ifndef PID_H_
#define PID_H_

#include <stdint.h>

#define DIFF_SCALE      4               // Differential scaling factor in average
#define LT_LENGTH    4                  // lastTemp array length
#define PID_TERM_MAX INT_FAST32_MAX/4   // Maximum p-, i-, d-term value to prevent overflow on sum

typedef struct pid_data {
    int_fast16_t lastTemp[LT_LENGTH];    // Storage for previous LT_LENGTH temperature values
    int_fast32_t errorSum;               // Integrator error sum
```

```
   int_fast16_t kP;                     // Proportional gain
   int_fast16_t kI;                     // Integral gain
   int_fast16_t kD;                     // Derivative gain
   int_fast32_t maxError;               // Maximum temperature error
   int_fast32_t maxErrorSum;            // Maximum integrator value
   int_fast32_t maxDeriv;               // Maximum derivative value
} pid_data_t;

void pid_init(int_fast16_t kp, int_fast16_t ki, int_fast16_t kd, struct pid_data *pid);

int_fast16_t pid_control(int_fast16_t setTemp, int_fast16_t sysTemp, struct pid_data *pid);

void pid_reset_integrator(struct pid_data *pid_st);



#endif /* PID_H_ */
```

pid.c

```
/*
Discrete Proportional-Integral-Derivative (PID)
controller implementation for thermal control system
Based on code from Atmel application note AVR221
See: http://www.atmel.com/images/doc2558.pdf?


*/

#include "AntennaController_v2_1.h"
#include <avr/pgmspace.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include "serial.h"

#include "pid.h"


// Initialize realtime PID data structure, calculate error limits
// Kp, Ki, Kd set gain for proportional, integral, & derivative terms respectively
// Imax sets maximum integral error to limit integral windup
void pid_init(int_fast16_t kp, int_fast16_t ki, int_fast16_t kd, struct pid_data *pid) {
   pid->errorSum = 0;
   for (int i = 0; i < sizeof(pid->lastTemp); i++) {
      pid->lastTemp[i] = 0;
   }

   pid->kP = kp;
   pid->kI = ki;
   pid->kD = kd;

   // Calculate limits to avoid integer overflow
   pid->maxError = (int_fast32_t)(PID_TERM_MAX/((int_fast32_t)pid->kP + 1));
   pid->maxErrorSum = (int_fast32_t)(PID_TERM_MAX/((int_fast32_t)pid->kI + 1));
   pid->maxDeriv = (int_fast32_t)(PID_TERM_MAX/((int_fast32_t)pid->kD +1));
}

// Workhorse function
// Call once per time interval to calculate new control output
// based on measured temperature & setpoint
int_fast16_t pid_control(int_fast16_t setTemp, int_fast16_t sysTemp, struct pid_data *pid) {
   int_fast16_t error;
   int_fast32_t p;
```

```
    int_fast32_t i;
    int_fast32_t d;
    int_fast32_t oldErrorSum;
    int_fast32_t newErrorSum;
    int_fast32_t output;

    // Calculate error signal
    error = setTemp - sysTemp;

    // Calculate proportional term
    // Limit to INT_FAST16_MAX to prevent overflow
    if(error > pid->maxError) {p = PID_TERM_MAX;}
    else if (error < -(pid->maxError)) {p = -PID_TERM_MAX;}
    else {p = (int_fast32_t)((int_fast32_t)(pid->kP) * (int_fast32_t)error);}

    // Save original error sum
    oldErrorSum = pid->errorSum;

    // Discrete integration of error
    newErrorSum = pid->errorSum + error;

    // Calculate integral term & update error sum
    // Limit based on calculated max error sum
    if (newErrorSum > pid->maxErrorSum) {
       i = PID_TERM_MAX;
       pid->errorSum = pid->maxErrorSum;
    }
    else if (newErrorSum < -(pid->maxErrorSum)) {
       i = -PID_TERM_MAX;
       pid->errorSum = -(pid->maxErrorSum);
    }
    else {
       i = (pid->kI) * newErrorSum;
       pid->errorSum = newErrorSum;
    }

    // Calculate derivative average
    int_fast32_t diff = pid->lastTemp[0] - sysTemp;
    //int_fast32_t diff = DIFF_SCALE * (pid->lastTemp[0] - sysTemp);
    //for (int i = LT_LENGTH-1; i > 0; i--) {
       //diff += DIFF_SCALE * (pid->lastTemp[i] - pid->lastTemp[i-1]);
       //}

    // Calculate d term from averaged derivative
    if (diff > pid->maxDeriv) {d = PID_TERM_MAX;}
    else if (diff < -(pid->maxDeriv)) {d = -PID_TERM_MAX;}
    else {d = (pid->kD) * diff;}

    // Shift lastTemp array
    for (int i = LT_LENGTH-1; i > 0; i--) {
       pid->lastTemp[i] = pid->lastTemp[i-1];
    }
    // Update 0th lastTemp
    pid->lastTemp[0] = sysTemp;

    // Calculate output & rescale
    output = p + i + d;
    //output = (p + i) + d/(DIFF_SCALE * LT_LENGTH);

    if (globalFlags.debug) {
       char err_char[12];
       char output_char[12];
       char p_char[12];
       char i_char[12];
```

```
        char d_char[12];

        itoa(error,err_char,10);
        sprintf_P(output_char, PSTR("%10li"), output);
        sprintf_P(p_char, PSTR("%10li"), p);
        sprintf_P(i_char, PSTR("%10li"), i);
        sprintf_P(d_char, PSTR("%10li"), d);

        send_str_P(PSTR("E:"));
        send_str(err_char);
        send_str_P(PSTR("\tP:"));
        send_str(p_char);
        send_str_P(PSTR("\tI:"));
        send_str(i_char);
        send_str_P(PSTR("\tD:"));
        send_str(d_char);
        send_str_P(PSTR("\tO:"));
        send_str(output_char);
        send_str_P(PSTR("\r\n"));
    }

    // Limit output to 16 bit integer to avoid overflow
    // Disable integrator to prevent windup during output saturation
    if (output > INT_FAST16_MAX) {
        output = INT_FAST16_MAX;
        pid->errorSum = oldErrorSum;  // Restore original error sum (undo integration)

    }
    else if (output <= -INT_FAST16_MAX) {
        output = -INT_FAST16_MAX;
        pid->errorSum = oldErrorSum;  // Restore original error sum (undo integration)
    }

    return (int_fast16_t)output;
}

// Integrator reset for when PID controller is turned off
void pid_reset_integrator(struct pid_data *pid_st) {
    pid_st->errorSum = 0;
}
```

A.2 Command Processing

A.2.1 Command Parsing Library

parse_cmd.h

```
#ifndef PARSE_CMD_H_
#define PARSE_CMD_H_

#include <stdint.h>

void setup_SerialCommand();

void serialCommand_readSerial();

char *serialCommand_next();

#endif /* PARSE_CMD_H_ */
```

parse_cmd.c

```
/*
Serial command parsing library based on ArduinoSerialCommand by Steven Cogswell and
```

```
modified by Stefan Rado.

See:
https://github.com/kroimon/Arduino-SerialCommand
https://github.com/scogswell/ArduinoSerialCommand

setup_SerialCommand() is where command handler functions in commands.c are registered
*/
#include <stdbool.h>

#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#include "AntennaController_v2_1.h"
#include "usb_serial.h"
#include "uart.h"
#include "commands.h"
#include "phased.h"

#define MAX_COMMAND_LENGTH 16
#define SERIALCOMMAND_BUFFER 64

/**********************************************************************/
/* Serial Command Parsing Functions                                   */
/**********************************************************************/

// Command/Handler List
struct SerialCommandCallback {
   uint8_t command[MAX_COMMAND_LENGTH+1];
   void (*function)();
   } *commandList;

//struct SerialCommandCallback *commandList;

uint8_t commandCount;

// Pointer to default (unmatched command) handler
void (*defaultHandler)(const char *);

char delim[2]; //Token delimiter character
char term;     //Command string terminator character

char buffer[SERIALCOMMAND_BUFFER+1];
uint8_t bufPos;
char *last;

void addCommand(const char *cmd, void (*function)()) {
   commandList = (struct SerialCommandCallback *)
               realloc(commandList, (commandCount + 1) * sizeof(struct SerialCommandCallback));
   strncpy(commandList[commandCount].command, cmd, MAX_COMMAND_LENGTH);
   commandList[commandCount].function = function;
   commandCount++;
}

void setDefaultHandler(void (*function)(const char *)) {
   defaultHandler = function;
}

void clearBuffer() {
   buffer[0] = '\0';
   bufPos = 0;
}

/*
```

```
The workhorse function
Reads characters in from the serial port, adds them to the buffer if printable
Matches to commands in commandList when terminator comes in
*/
void serialCommand_readSerial() {
#ifdef USB_SERIAL
   while (usb_serial_available() > 0) {
      char inChar = usb_serial_getchar();
#else
   while (uart_available() > 0) {
      char inChar = uart_getchar();
#endif
      if (inChar == term) {
         char *command = strtok_r(buffer, delim, &last);
         if (command != NULL) {
            bool matched = false;
            for (int i = 0; i < commandCount; i++) {
               if (strncasecmp(command, commandList[i].command, MAX_COMMAND_LENGTH)==0) {
                  (*commandList[i].function)();
                  matched = true;
                  break;
               }
            }
            if (!matched && (defaultHandler != NULL)) {
               (*defaultHandler)(command);
            }
         }
         clearBuffer();
      }
      else if (isprint(inChar)) {

         if (bufPos < SERIALCOMMAND_BUFFER) {
            buffer[bufPos++] = inChar;
            buffer[bufPos] = '\0';
         }
         else {
         }
      }
   }
}

//Return next token (argument) from buffer
char *serialCommand_next() {
   return strtok_r(NULL, delim, &last);
}

// All commands are registered here with the command string & handler function name
// Handler functions go in commands.c/commands.h
void setup_SerialCommand() {
   commandList = NULL;
   commandCount = 0;
   defaultHandler = NULL;
   strcpy(delim, " ");  // Command/Argument & Argument/Argument delimiter
   term = '\r';      // Command/Argument string terminator
   *last = NULL;

   addCommand("SETDAC", set_dac_output);
   addCommand("SETBEAM", set_beam_angle);

   addCommand("SETSENSE", set_sense);
   addCommand("SETTEMP", set_temp);
   addCommand("SETMODE", set_mode);
   addCommand("SETTEC", set_tec_manual);
   addCommand("SETPID", set_pid);
```

```
    addCommand("SETPUMP", set_pump_modes);
    addCommand("SETPUMPPARAM", set_pump_params);

    addCommand("SETVALVE", set_valves);
    addCommand("SETSERVOPW", set_servo_pw);
    addCommand("SETVALVEPARAM", set_valve_params);

    addCommand("GETSENSE", get_sense);
    addCommand("GETTEMP", get_temp);
    addCommand("GETMODE", get_mode);
    addCommand("GETPID", get_pid);
    addCommand("GETTEC", get_tec_status);
    addCommand("GETPUMP", get_pumps);
    addCommand("GETVALVE", get_valve_params);

    addCommand("SCRAM", set_scram);

    addCommand("READY?", ready);

    addCommand("dbg", debug);

    setDefaultHandler(error);
}
```

A.2.2 Command Execution Handler Functions

commands.h

```
/*
 * commands.h
 *
 * Created: 4/11/2014 12:40:22 AM
 *  Author: Nick
 */


#ifndef COMMANDS_H_
#define COMMANDS_H_

void set_sense();
void set_temp();
void set_mode();
void set_tec_manual();
void set_pid();
void set_pump_modes();
void set_pump_params();

void set_valves();
void set_valve_params();
void set_servo_pw();

void get_sense();
void get_temp();
void get_mode();
void get_pid();
void get_tec_status();
void get_pumps();
void get_valve_params();

void set_scram();
void ready();
void debug();

void error(const char *cmd);
```

```
#endif /* COMMANDS_H_ */
```

commands.c

```
/* Handler functions for serial commands */

#include <avr/pgmspace.h>
#include <avr/eeprom.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include "commands.h"
#include "AntennaController_v2_1.h"
#include "pwm.h"
#include "pid.h"
#include "serial.h"
#include "parse_cmd.h"
#include "AD5668.h"

#define OK     send_str_P(PSTR("OK\r\n"))
#define ERROR  send_str_P(PSTR("ERROR\r\n"))

/*************************************************************************/
/* Set Commands                                                          */
/*************************************************************************/

// Set sense temperature source for PID controller
void set_sense() {
   char *arg = serialCommand_next();
   if (arg != NULL) {
      if    (strcasecmp_P(arg, PSTR("TEC"))==0)    {myTempSenseSource = TEC;}
      else if (strcasecmp_P(arg, PSTR("MODULE"))==0)  {myTempSenseSource = MODULE;}
      else {
         ERROR;
         return;
      }
      OK;
      return;
   }
   else {
      ERROR;
      return;
   }
}

// Set target (desired) temperature
void set_temp() {
   char *arg = serialCommand_next();
   if (arg != NULL) {
      int_fast16_t temp = (int_fast16_t)(10.0*strtod(arg, NULL));
      if ((temp > MIN_SET_TEMP)&&(temp < MAX_SET_TEMP)) {
         t_tgt = temp;
         OK;
         return;
      }
      else {
         ERROR;
         return;
      }
   }
   else {
      ERROR;
```

```
            return;
        }
    }

    // Set temperature control mode (automatic/off)
    void set_mode()    {
        char *arg = serialCommand_next();
        if (arg != NULL) {
            if (strcasecmp_P(arg, PSTR("AUTO"))==0) {
                globalFlags.auto_mode = 1;
            }
            else if (strcasecmp_P(arg, PSTR("MAN"))==0) {
                globalFlags.auto_mode = 0;
            }
            else if (strcasecmp_P(arg, PSTR("OFF"))==0) {
                globalFlags.auto_mode = 0;
                set_tec(TEC_OFF, 0);
            }
            else {
                ERROR;
                return;
            }
            OK;
            return;
        }
        else {
            ERROR;
            return;
        }
    }

    void set_tec_manual() {
        char *arg1 = serialCommand_next();
        char *arg2 = serialCommand_next();
        if ((arg1 != NULL)&&(arg2 != NULL)) {
            int_fast16_t pw = strtol(arg2, NULL, 10);
            if ((pw > -1)&&(pw < 1024)) {
                if (strcasecmp_P(arg1, PSTR("HEAT"))==0) {
                    set_tec(TEC_HEAT, pw);
                }
                else if (strcasecmp_P(arg1, PSTR("COOL"))==0) {
                    set_tec(TEC_COOL, pw);
                }
                else {
                    ERROR;
                    return;
                }
                OK;
                return;
            }
            else {
                ERROR;
                return;
            }
        }
        else {
            ERROR;
            return;
        }
    }

    // Set temp control PID parameters
    void set_pid() {
        char *arg1 = serialCommand_next();  // KP/KI/KD/IMAX argument
```

```c
   char *arg2 = serialCommand_next();  // Float/Int gain argument

   if ((arg1 != NULL)&&(arg2 != NULL)) {
      int_fast32_t tmp = strtol(arg2, NULL, 10);
      if (tmp > INT_FAST16_MAX) {
         ERROR;
         return;
      }
      else {
         if    (strcasecmp_P(arg1, PSTR("KP"))==0) {
            myPIDparams.k_p = (int_fast16_t)tmp;
         }
         else if (strcasecmp_P(arg1, PSTR("KI"))==0) {
            myPIDparams.k_i = (int_fast16_t)tmp;
         }
         else if (strcasecmp_P(arg1, PSTR("KD"))==0) {
            myPIDparams.k_d = (int_fast16_t)tmp;
         }
         else {
            ERROR;
            return;
         }
      }

      pid_init(myPIDparams.k_p, myPIDparams.k_i, myPIDparams.k_d, &myPIDdata);
      OK;
      return;
   }
   else if (strcasecmp_P(arg1, PSTR("WRITE"))==0) {
      eeprom_update_block(&myPIDparams, EEP_ADDRESS_PID, sizeof(myPIDparams));
      OK;
      return;
   }
   else {
      ERROR;
      return;
   }
}

// Set pump modes
// SETPUMP 1,2,3 IN,OUT,OFF -- Sets pumps 1 2 & 3 to IN, OUT, & OFF respectively
void set_pump_modes() {
   char *arg1 = serialCommand_next();
   char *arg2 = serialCommand_next();
   char *last1, *last2;

   char *pmp = strtok_r(arg1, ",", &last1);
   char *dir = strtok_r(arg2, ",", &last2);
   if ((pmp==NULL)||(dir==NULL)) {
      ERROR;
      return;
   }
   else {
      enum pump pnum = PUMP_NULL;
      if    (strcasecmp_P(pmp, PSTR("1"))==0) {pnum = PUMP1;}
      else if (strcasecmp_P(pmp, PSTR("2"))==0) {pnum = PUMP2;}
      else if (strcasecmp_P(pmp, PSTR("3"))==0) {pnum = PUMP3;}
      else if (strcasecmp_P(pmp, PSTR("4"))==0) {pnum = PUMP4;}
      else {
         ERROR;
         return;
      }
      if    (strcasecmp_P(dir, PSTR("IN"))==0) {
         set_pump(pnum, MODE_IN, myPumpParams[pnum-1].pw);
```

```c
      }
      else if (strcasecmp_P(dir, PSTR("OUT"))==0) {
         set_pump(pnum, MODE_OUT, myPumpParams[pnum-1].pw);
      }
      else if (strcasecmp_P(dir, PSTR("OFF"))==0) {
         set_pump(pnum, MODE_OFF, myPumpParams[pnum-1].pw);
      }
      else {
         ERROR;
         return;
      }
      OK;
      for (uint8_t i = 1; i < 4; i++) {
         char *pmp = strtok_r(NULL, ",", &last1);
         char *dir = strtok_r(NULL, ",", &last2);
         enum pump pnum = PUMP_NULL;
         if    ((pmp == NULL)||dir==NULL) {return;}
         else if  (strcasecmp_P(pmp, PSTR("1"))==0) {pnum = PUMP1;}
         else if (strcasecmp_P(pmp, PSTR("2"))==0) {pnum = PUMP2;}
         else if (strcasecmp_P(pmp, PSTR("3"))==0) {pnum = PUMP3;}
         else if (strcasecmp_P(pmp, PSTR("4"))==0) {pnum = PUMP4;}
         else {
            ERROR;
            return;
         }
         if    (strcasecmp_P(dir, PSTR("IN"))==0) {
            set_pump(pnum, MODE_IN, myPumpParams[pnum-1].pw);
         }
         else if (strcasecmp_P(dir, PSTR("OUT"))==0) {
            set_pump(pnum, MODE_OUT, myPumpParams[pnum-1].pw);
         }
         else if (strcasecmp_P(dir, PSTR("OFF"))==0) {
            set_pump(pnum, MODE_OFF, myPumpParams[pnum-1].pw);
         }
         else {
            ERROR;
            return;
         }
      }
   }
}

// Set pump parameters
void set_pump_params() {
   char *arg1 = serialCommand_next(); // Pump number / WRITE
   char *arg2 = serialCommand_next(); // Pump parameter
   char *arg3 = serialCommand_next(); // Parameter value

   int_fast16_t tmp = strtol(arg3, NULL, 10);

   if ((arg1 != NULL)&&(arg2 != NULL)&&(arg3 != NULL)) {
      enum pump pnum = PUMP_NULL;
      if    (strcasecmp_P(arg1, PSTR("1"))==0) {pnum = PUMP1;}
      else if (strcasecmp_P(arg1, PSTR("2"))==0) {pnum = PUMP2;}
      else if (strcasecmp_P(arg1, PSTR("3"))==0) {pnum = PUMP3;}
      else if (strcasecmp_P(arg1, PSTR("4"))==0) {pnum = PUMP4;}
      else {
         ERROR;
         return;
      }

      if (strcasecmp_P(arg2, PSTR("SPEED"))==0) {
         if ((tmp < 0)||(tmp > 1023)) {
            ERROR;
```

```
            return;
         }
         else {
            myPumpParams[pnum-1].pw = tmp;
            set_pump(pnum, get_pump(pnum).mode, tmp);
            OK;
            return;
         }
      }
      else if (strcasecmp_P(arg2, PSTR("DELAY"))==0) {
         myPumpParams[pnum-1].timer_ms = tmp;
         OK;
         return;
      }
      else {
         ERROR;
         return;
      }
   }
   else if (arg1 != NULL) {
      if (strcasecmp_P(arg1, PSTR("WRITE"))==0) {
         eeprom_update_block(myPumpParams, EEP_ADDRESS_PUMP, sizeof(myPumpParams));
         OK;
         return;
      }
      else {
         ERROR;
         return;
      }
   }
   else {
      ERROR;
      return;
   }
}

// Set all valves to desired positions
// "SETVALVE x,y,z a,b,c" -> Sets valves x,y,z... to positions a,b,c...
void set_valves() {
   char *arg1 = serialCommand_next();
   char *arg2 = serialCommand_next();
   char *last1, *last2;


   char *vlv = strtok_r(arg1, ",", &last1);
   char *pos = strtok_r(arg2, ",", &last2);
   if ((vlv==NULL)||(pos==NULL)) {
      ERROR;
      return;
   }
   else {
      int valve = strtol(vlv, NULL, 10)-1;
      OK;
      if    (strcasecmp_P(pos, PSTR("0"))==0) {
         set_servo(valve,myValveParams[0].pos0);
      }
      else if (strcasecmp_P(pos, PSTR("1"))==0) {
         set_servo(valve,myValveParams[0].pos1);
      }
      else if (strcasecmp_P(pos, PSTR("2"))==0) {
         set_servo(valve,myValveParams[0].pos2);
      }
      else {
         ERROR;
```

```
            return;
        }
        for (uint8_t i = 1; i < 8; i++) {
            char *vlv = strtok_r(NULL, ",", &last1);
            char *pos = strtok_r(NULL, ",", &last2);
            int valve = strtol(vlv, NULL, 10)-1;
            if    (strcasecmp_P(pos, PSTR("0"))==0) {
                set_servo(valve,myValveParams[i].pos0);
            }
            else if (strcasecmp_P(pos, PSTR("1"))==0) {
                set_servo(valve,myValveParams[i].pos1);
            }
            else if (strcasecmp_P(pos, PSTR("2"))==0) {
                set_servo(valve,myValveParams[i].pos2);
            }
            else {
                ERROR;
                return;
            }
        }
    }
}

// Set valves' position<>pulse width mapping
// "SETVALVEPARAM n a,b,c" -> Sets valve n's Off,A,B pulse widths
// "SETVALVEPARAM WRITE" -> Writes valve parameters to EEPROM
void set_valve_params() {
    char *arg1 = serialCommand_next();
    char *arg2 = serialCommand_next();
    if (strcasecmp_P(arg1, PSTR("WRITE"))==0) {
        eeprom_update_block(&myValveParams, EEP_ADDRESS_VALVE, sizeof(myValveParams));
        OK;
        return;
    }
    else {
        char *last;
        char *pw1_str = strtok_r(arg2, ",", &last);
        char *pw2_str = strtok_r(NULL, ",", &last);
        char *pw3_str = strtok_r(NULL, ",", &last);

        if ((pw1_str!=NULL)&&(pw2_str!=NULL)&&(pw3_str!=NULL)) {
            uint8_t pw[3];
            uint8_t vnum;
            pw[0] = strtol(pw1_str, NULL, 10);
            pw[1] = strtol(pw2_str, NULL, 10);
            pw[2] = strtol(pw3_str, NULL, 10);
            for (int i = 0; i < 3; i++) {
                if ((pw[i] < 0)||(pw[i] > 255))  {
                    ERROR;
                    return;
                }
            }

            if    (strcasecmp_P(arg1, PSTR("1"))==0) {vnum = 0;}
            else if (strcasecmp_P(arg1, PSTR("2"))==0) {vnum = 1;}
            else if (strcasecmp_P(arg1, PSTR("3"))==0) {vnum = 2;}
            else if (strcasecmp_P(arg1, PSTR("4"))==0) {vnum = 3;}
            else if (strcasecmp_P(arg1, PSTR("5"))==0) {vnum = 4;}
            else if (strcasecmp_P(arg1, PSTR("6"))==0) {vnum = 5;}
            else if (strcasecmp_P(arg1, PSTR("7"))==0) {vnum = 6;}
            else if (strcasecmp_P(arg1, PSTR("8"))==0) {vnum = 7;}
            else {
                ERROR;
                return;
```

```
      }
      myValveParams[vnum].pos0 = pw[0];
      myValveParams[vnum].pos1 = pw[1];
      myValveParams[vnum].pos2 = pw[2];
      OK;
      return;
    }
    else {
      ERROR;
      return;
    }
  }
}

// Set servo valve pulse width directly
// "SETSERVO x y" -> Sets servo x to pw y
void set_servo_pw() {
  char *arg1 = serialCommand_next();
  char *arg2 = serialCommand_next();

  enum servo myservo;

  if    (strcasecmp_P(arg1, PSTR("1"))==0) myservo = SERVO1;
  else if (strcasecmp_P(arg1, PSTR("2"))==0) myservo = SERVO2;
  else if (strcasecmp_P(arg1, PSTR("3"))==0) myservo = SERVO3;
  else if (strcasecmp_P(arg1, PSTR("4"))==0) myservo = SERVO4;
  else if (strcasecmp_P(arg1, PSTR("5"))==0) myservo = SERVO5;
  else if (strcasecmp_P(arg1, PSTR("6"))==0) myservo = SERVO6;
  else if (strcasecmp_P(arg1, PSTR("7"))==0) myservo = SERVO7;
  else if (strcasecmp_P(arg1, PSTR("8"))==0) myservo = SERVO8;
  else {
    ERROR;
    return;
  }

  if (arg2 != NULL) {
    uint8_t mypwm = (uint8_t)strtol(arg2, NULL, 10);
    if ((mypwm > -1)&&(mypwm < 1024)) {
      set_servo(myservo,mypwm);
      OK;
      return;
    }
    else {
      ERROR;
      return;
    }
  }
  else {
    ERROR;
    return;
  }
}


/**************************************************************************/
/* Get Commands                                                           */
/**************************************************************************/

// Function to print temperature stored as 10*temp in 16-bit integer to serial port
void print_temp(int_fast16_t temp) {
  char temp_str[8];
  sprintf_P(temp_str,PSTR("%.1f"),(float)temp/10);
  send_str(temp_str);
}
```

```
// Get PID sense temperature source
void get_sense() {
   switch (myTempSenseSource) {
      case TEC:
         send_str_P(PSTR("TEC\r\n"));
         break;
      case MODULE:
         send_str_P(PSTR("MODULE\r\n"));
         break;
      default:
         break;
   }
}

// Get temperature from one thermistor/setpoint or all at once
void get_temp()   {
   char *arg = serialCommand_next();
   char *output;
   if (arg != NULL) {
      if    (strcasecmp_P(arg, PSTR("TEC"))==0) {
         print_temp(t_tec);
         send_str_P(PSTR("\r\n"));
         return;
      }
      else if (strcasecmp_P(arg, PSTR("MOD"))==0) {
         print_temp(t_mod);
         send_str_P(PSTR("\r\n"));
         return;
      }
      else if (strcasecmp_P(arg, PSTR("SINK"))==0) {
         print_temp(t_sink);
         send_str_P(PSTR("\r\n"));
         return;
      }
      else if (strcasecmp_P(arg, PSTR("TGT"))==0) {
         print_temp(t_tgt);
         send_str_P(PSTR("\r\n"));
         return;
      }
      else {
         ERROR;
         return;
      }
   }
   else {
      send_str_P(PSTR("TEC:"));
      print_temp(t_tec);
      send_str_P(PSTR(",MOD:"));
      print_temp(t_mod);
      send_str_P(PSTR(",SINK:"));
      print_temp(t_sink);
      send_str_P(PSTR(",TGT:"));
      print_temp(t_tgt);
      send_str_P(PSTR("\r\n"));
      return;
   }
}

// Get temperature control mode
void get_mode()   {
   struct tec_status myTECstatus = get_tec();
   if (globalFlags.in_scram) {
      send_str_P(PSTR("SCRAM\r\n"));
```

```
        return;
    }
    else if  (globalFlags.auto_mode) {
        send_str_P(PSTR("AUTO\r\n"));
        return;
    }
    else if (!globalFlags.auto_mode) {
        if (myTECstatus.mode == TEC_OFF) {
            send_str_P(PSTR("OFF\r\n"));
        }
        else {
            send_str_P(PSTR("MANUAL\r\n"));
        }
        return;
    }
}

// Get temperature control PID parameters
void get_pid() {
    char kpbuf[10], kibuf[10], kdbuf[10], maxerrbuf[10], maxerrsumbuf[10];
    sprintf_P(kpbuf,PSTR("%i"),myPIDdata.kP);
    sprintf_P(kibuf,PSTR("%i"),myPIDdata.kI);
    sprintf_P(kdbuf,PSTR("%i"),myPIDdata.kD);
    sprintf_P(maxerrbuf,PSTR("%li"),myPIDdata.maxError);
    sprintf_P(maxerrsumbuf,PSTR("%li"),myPIDdata.maxErrorSum);

    send_str_P(PSTR("KP:"));
    send_str(kpbuf);
    send_str_P(PSTR(",KI:"));
    send_str(kibuf);
    send_str_P(PSTR(",KD:"));
    send_str(kdbuf);
    send_str_P(PSTR(",ERRMAX:"));
    send_str(maxerrbuf);
    send_str_P(PSTR(",SUMMAX:"));
    send_str(maxerrsumbuf);
    send_str_P(PSTR("\r\n"));

}

void get_tec_status() {
    struct tec_status myTECstatus = get_tec();
    char pw_char[8];
    char errorsum_char[15];
    sprintf_P(pw_char, PSTR("%u"), myTECstatus.pw);
    sprintf_P(errorsum_char, PSTR("%li"), myPIDdata.errorSum);

    switch (myTECstatus.mode) {
        case TEC_OFF:
            send_str_P(PSTR("OFF"));
            break;
        case TEC_HEAT:
            send_str_P(PSTR("HEAT"));
            break;
        case TEC_COOL:
            send_str_P(PSTR("COOL"));
            break;
        default:
            break;
    }
    send_str_P(PSTR(",PW:"));
    send_str(pw_char);
    send_str_P(PSTR(",INT:"));
    send_str(errorsum_char);
```

```
    send_str_P(PSTR("\r\n"));
}

void get_pumps() {      // Get pump parameters
    struct pump_status myPumpStatus[4];
    for (int i = 1; i < 5; i++) {
        myPumpStatus[i-1] = get_pump(i);
    }
    send_str_P(PSTR("PUMP:\tMODE:\tASPD:\tSSPD:\tDELAY:\r\n"));
    for (int i = 0; i < 4; i++) {
        char istr[2];
        sprintf_P(istr, PSTR("%i"),i+1);
        send_str_P(PSTR("\t"));
        send_str(istr);
        send_str_P(PSTR("\t"));
        if    (myPumpStatus[i].mode==MODE_OFF) {send_str_P(PSTR("OFF"));}
        else if (myPumpStatus[i].mode==MODE_OUT) {send_str_P(PSTR("OUT"));}
        else if (myPumpStatus[i].mode==MODE_IN) {send_str_P(PSTR("IN"));}
        send_str_P(PSTR("\t"));
        char apw[8], spw[8], delay[8];
        sprintf_P(apw, PSTR("%i"),myPumpStatus[i].pw);
        send_str(apw);
        send_str_P(PSTR("\t"));
        sprintf_P(spw, PSTR("%i"),myPumpParams[i].pw);
        send_str(spw);
        send_str_P(PSTR("\t"));
        sprintf_P(delay, PSTR("%i"),myPumpParams[i].timer_ms);
        send_str(delay);
        send_str_P(PSTR("\r\n"));
    }
}

// Get pulse widths for valve positions
void get_valve_params() {
    send_str_P(PSTR("Valve/Pos:\t0\t1\t2\r\n"));
    for (uint8_t i = 0; i < 8; i++) {
        char istr[2];
        sprintf_P(istr,PSTR("%i"),i+1);
        send_str_P(PSTR("\t"));
        send_str(istr);

        char pw0[8], pw1[8], pw2[8];
        sprintf_P(pw0,PSTR("%i"),myValveParams[i].pos0);
        send_str_P(PSTR("\t"));
        send_str(pw0);
        sprintf_P(pw1,PSTR("%i"),myValveParams[i].pos1);
        send_str_P(PSTR("\t"));
        send_str(pw1);
        sprintf_P(pw2,PSTR("%i"),myValveParams[i].pos2);
        send_str_P(PSTR("\t"));
        send_str(pw2);

        send_str_P(PSTR("\r\n"));
    }
}

void set_scram() {   // SCRAM power outputs
    char *arg = serialCommand_next();
    if (arg != NULL) {
        if (strcasecmp_P(arg, PSTR("OFF"))==0) {
            globalFlags.in_scram = 0;
            OK;
            return;
        }
```

```
         else {
            ERROR;
            return;
         }
      }
      else {
         scram();
         OK;
         return;
      }
}

void ready() { // Respond to "READY?" polling
   send_str_P(PSTR("READY\r\n"));
   return;
}

void debug() {
   char *arg = serialCommand_next();
   if (strcasecmp_P(arg,PSTR("ON"))==0) {
      globalFlags.debug = 1;
   }
   else if (strcasecmp_P(arg,PSTR("OFF"))==0) {
      globalFlags.debug = 0;
   }
   else {
      ERROR;
      return;
   }
   OK;
}

//Respond to unrecognized command
void error(const char *cmd) {
   ERROR;
}
```

phased.h

```
/*
 * phased.h
 *
 * Created: 5/28/2014 3:44:48 PM
 *  Author: Nick
 */


#ifndef PHASED_H_
#define PHASED_H_

void set_dac_output();

void set_beam_angle();

#endif /* PHASED_H_ */
```

phased.c

```
/*
 * phased.c
 *
 * Created: 5/28/2014 3:44:39 PM
 *  Author: Nick
```

```c
 */
#include <avr/pgmspace.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#include "AntennaController_v2_1.h"
#include "parse_cmd.h"
#include "serial.h"
#include "phased.h"
#include "AD5668.h"

#define FREQ      2.46e9      // Operating frequency (Hz)
#define LT_SPEED  2.9979e8 // Speed of light (m/s)
#define DX        0.0737      // Element separation in X (meters)
#define DY        0.0426      // Element separation in Y (meters)
#define PHA_STATIC   90       // Static phase delay ()

#define WORD_V    5300     // DAC word / voltage

#define OK     send_str_P(PSTR("OK\r\n"))
#define ERROR  send_str_P(PSTR("ERROR\r\n"))

void set_dac_output() {
   enum dac_add myDac = DAC_A;
   char *arg1 = serialCommand_next();
   char *arg2 = serialCommand_next();
   if ((arg1!=NULL)&&(arg2!=NULL)) {
      if (strcasecmp_P(arg1, PSTR("1"))==0) {myDac = DAC_A;}
      else if (strcasecmp_P(arg1, PSTR("2"))==0) {myDac = DAC_B;}
      else if (strcasecmp_P(arg1, PSTR("3"))==0) {myDac = DAC_C;}
      else if (strcasecmp_P(arg1, PSTR("4"))==0) {myDac = DAC_D;}
      else if (strcasecmp_P(arg1, PSTR("5"))==0) {myDac = DAC_E;}
      else if (strcasecmp_P(arg1, PSTR("6"))==0) {myDac = DAC_F;}
      else if (strcasecmp_P(arg1, PSTR("7"))==0) {myDac = DAC_G;}
      else if (strcasecmp_P(arg1, PSTR("8"))==0) {myDac = DAC_H;}
      else if (strcasecmp_P(arg1, PSTR("ALL"))==0) {myDac = DAC_ALL;}
      else {
         ERROR;
         return;
      }
      uint_fast16_t out = strtol(arg2, NULL, 10);

      set_dac_word(myDac,out);
      OK;
   }
   else {
      ERROR;
      return;
   }
}

void set_beam_angle() {
   char *arg1 = serialCommand_next();
   char *arg2 = serialCommand_next();

   if ((arg1!=NULL)&&(arg2!=NULL)) {
      // Get azimuth & elevation scan angles in degrees (conv to rad)
      double az = 1.25*strtod(arg1, NULL)*(M_PI/180.0);
      double el = 1.25*strtod(arg2, NULL)*(M_PI/180.0);

      // Calculate progressive phase shifts in x & y in degrees
      double beta_x = (180.0/M_PI)*((2.0*M_PI*FREQ)/LT_SPEED)*DX*sinf(fabs(az));
```

```
    double beta_y = (180.0/M_PI)*((2.0*M_PI*FREQ)/LT_SPEED)*DY*sinf(fabs(el));

    double phase[7] =
       {PHA_STATIC,PHA_STATIC,PHA_STATIC,PHA_STATIC,PHA_STATIC,PHA_STATIC,PHA_STATIC};

    if (az != 0) {
       phase[0] += beta_x;
       phase[1] += beta_x;
       phase[4] += beta_x;
    }
    if (el != 0) {
       phase[0] += 2.0*beta_y;
    }

    if (az < 0) {
       phase[2] += 2.0*beta_x;
       phase[3] += 2.0*beta_x;
    }
    else if (az > 0) {
       phase[5] += 2.0*beta_x;
       phase[6] += 2.0*beta_x;
    }

    if (el < 0) {
       phase[3] += beta_y;
       phase[5] += beta_y;

       phase[2] += 3.0*beta_y;
       phase[6] += 3.0*beta_y;

       phase[1] += 4.0*beta_y;
    }
    else if (el > 0) {
       phase[2] += beta_y;
       phase[6] += beta_y;

       phase[3] += 3.0*beta_y;
       phase[5] += 3.0*beta_y;

       phase[4] += 4.0*beta_y;
    }

    for (int i = 0; i < 7; i++) {
       phase[i] = fmodf(phase[i],360.0);    // Phase shifts modulo 360
    }

    int_fast16_t ph_word[7] = {0,0,0,0,0,0,0};

    for (int i = 0; i < 7; i++) {
       double out_volt =
          2e-9*powf(phase[i],3.0) + 3e-5*powf(phase[i],2) + 1.18e-2*phase[i] + 2.16e-2;
       if ((out_volt >= 0)&&(out_volt <= 1.8)) {
          out_volt -= 0.12;
       }
       if (out_volt  >= 2.5) {
          out_volt += 0.05;
       }
       ph_word[i] = (int_fast16_t)floorf(WORD_V*out_volt);
    }

    enum dac_add myDacs[7] = {DAC_A, DAC_B, DAC_C, DAC_D, DAC_E, DAC_F, DAC_G};

    for (int i = 0; i < 7; i++) {
       set_dac_word(myDacs[i],ph_word[i]);
```

```
        }
        OK;
    }
    else {
        ERROR;
        return;
    }
}
```

A.3 Peripheral Control Code

A.3.1 Analog/Digital Conversion & Temperature Conversion

adc.h

```
/*
 * adc.h
 *
 * Created: 11/21/2013 5:50:09 PM
 *  Author: Nick
 */


#ifndef ADC_H_
#define ADC_H_

#include <stdint.h>
#include <avr/io.h>

void configure_adc(void);

int_fast16_t read_adc(uint8_t pin);



#endif /* ADC_H_ */
```

adc.c

```
/*
ADC configuration & read functions
*/

#include <stdint.h>
#include <avr/io.h>

void configure_adc(void) {
    // Configure ADC clock & reference

    ADCSRA |= ((1 << ADPS2)|(1 << ADPS1)|(1 << ADPS0)); // Set ADC clock to 16MHz/128=125kHz
    ADMUX &= ~((1<<REFS1)|(1<<REFS0));  // Set ADC reference to AREF (4.096V input)

    ADMUX |= (1<<REFS0);

    ADCSRA |= (1<<ADEN); // Enable ADC

    // Set up Timer 0 to generate TIMER0_OVF_vect at 100 Hz

    TCCR0A |= ((1<<WGM01)|(1<<WGM00));
    TCCR0B |= (1<<WGM02);   // Set Fast PWM Mode

    TCCR0B &= ~(1<<CS02);   // Set clock to 16MHz/64
    TCCR0B |= (1<<CS01);
```

```
    TCCR0B |= (1<<CS00);

    OCR0A = 249;   // Set TOP to 249 -> TOV rate = 16MHz/(64*(249+1)) = 1000 Hz

    TIMSK0 |= (1<<TOIE0);   // Enable Timer 0 Overflow Interrupt
}

int_fast16_t read_adc(uint8_t pin) {
    pin = (pin & 0x07);              // Mask out only MUX2...MUX0 bits
    ADMUX = ((ADMUX & 0xF8) | pin);  // Select pin
    ADCSRA |= (1<<ADSC);             // Start ADC conversion
    while (ADCSRA & (1<<ADSC));      // Wait for conversion to complete
    return (ADC);
}
```

adc2temp.h

```
#ifndef ADC2TEMP_H_
#define ADC2TEMP_H_


//adc2temp Header File

#include <stdint.h>

int_fast16_t adc2temp(int_fast16_t temp);



#endif /* ADC2TEMP_H_ */
```

adc2temp.c

```
/*
LUT for Vishay 01M1002KF NTC Thermistor on 4.096V reference w/ 10k low-side resistor
Converts ADC value to signed 16 bit int representing 10*[temp in deg C]
0.1C resolution for 0.4-39C
0.2C resolution for -22.1-67.3C
1.0C resolution for -51.2-120.1C
*/

#include "adc2temp.h"
#include <avr/pgmspace.h>
#include <stdint.h>

static const int_fast16_t tempcnv[1024] PROGMEM = {
    0xF555, 0xFD07, 0xFD37, 0xFD5A, 0xFD76, 0xFD8D, 0xFDA1, 0xFDB3,
    0xFDC3, 0xFDD1, 0xFDDE, 0xFDEA, 0xFDF5, 0xFE00, 0xFE09, 0xFE13,
    0xFE1B, 0xFE24, 0xFE2B, 0xFE33, 0xFE3A, 0xFE41, 0xFE48, 0xFE4E,
    0xFE54, 0xFE5A, 0xFE60, 0xFE66, 0xFE6B, 0xFE70, 0xFE75, 0xFE7A,
    0xFE7F, 0xFE84, 0xFE89, 0xFE8D, 0xFE91, 0xFE96, 0xFE9A, 0xFE9E,
    0xFEA2, 0xFEA6, 0xFEAA, 0xFEAE, 0xFEB1, 0xFEB5, 0xFEB9, 0xFEBC,
    0xFEBF, 0xFEC3, 0xFEC6, 0xFEC9, 0xFECD, 0xFED0, 0xFED3, 0xFED6,
    0xFED9, 0xFEDC, 0xFEDF, 0xFEE2, 0xFEE5, 0xFEE8, 0xFEEA, 0xFEED,
    0xFEF0, 0xFEF3, 0xFEF5, 0xFEF8, 0xFEFB, 0xFEFD, 0xFF00, 0xFF02,
    0xFF05, 0xFF07, 0xFF0A, 0xFF0C, 0xFF0E, 0xFF11, 0xFF13, 0xFF15,
    0xFF18, 0xFF1A, 0xFF1C, 0xFF1E, 0xFF20, 0xFF23, 0xFF25, 0xFF27,
    0xFF29, 0xFF2B, 0xFF2D, 0xFF2F, 0xFF31, 0xFF33, 0xFF35, 0xFF37,
    0xFF39, 0xFF3B, 0xFF3D, 0xFF3F, 0xFF41, 0xFF43, 0xFF45, 0xFF47,
    0xFF49, 0xFF4A, 0xFF4C, 0xFF4E, 0xFF50, 0xFF52, 0xFF54, 0xFF55,
    0xFF57, 0xFF59, 0xFF5B, 0xFF5C, 0xFF5E, 0xFF60, 0xFF61, 0xFF63,
    0xFF65, 0xFF66, 0xFF68, 0xFF6A, 0xFF6B, 0xFF6D, 0xFF6F, 0xFF70,
    0xFF72, 0xFF73, 0xFF75, 0xFF77, 0xFF78, 0xFF7A, 0xFF7B, 0xFF7D,
```

```
0xFF7E, 0xFF80, 0xFF81, 0xFF83, 0xFF84, 0xFF86, 0xFF87, 0xFF89,
0xFF8A, 0xFF8C, 0xFF8D, 0xFF8F, 0xFF90, 0xFF91, 0xFF93, 0xFF94,
0xFF96, 0xFF97, 0xFF99, 0xFF9A, 0xFF9B, 0xFF9D, 0xFF9E, 0xFF9F,
0xFFA1, 0xFFA2, 0xFFA4, 0xFFA5, 0xFFA6, 0xFFA8, 0xFFA9, 0xFFAA,
0xFFAC, 0xFFAD, 0xFFAE, 0xFFAF, 0xFFB1, 0xFFB2, 0xFFB3, 0xFFB5,
0xFFB6, 0xFFB7, 0xFFB9, 0xFFBA, 0xFFBB, 0xFFBC, 0xFFBE, 0xFFBF,
0xFFC0, 0xFFC1, 0xFFC3, 0xFFC4, 0xFFC5, 0xFFC6, 0xFFC7, 0xFFC9,
0xFFCA, 0xFFCB, 0xFFCC, 0xFFCE, 0xFFCF, 0xFFD0, 0xFFD1, 0xFFD2,
0xFFD3, 0xFFD5, 0xFFD6, 0xFFD7, 0xFFD8, 0xFFD9, 0xFFDB, 0xFFDC,
0xFFDD, 0xFFDE, 0xFFDF, 0xFFE0, 0xFFE1, 0xFFE3, 0xFFE4, 0xFFE5,
0xFFE6, 0xFFE7, 0xFFE8, 0xFFE9, 0xFFEA, 0xFFEC, 0xFFED, 0xFFEE,
0xFFEF, 0xFFF0, 0xFFF1, 0xFFF2, 0xFFF3, 0xFFF4, 0xFFF6, 0xFFF7,
0xFFF8, 0xFFF9, 0xFFFA, 0xFFFB, 0xFFFC, 0xFFFD, 0xFFFE, 0xFFFF,
0x0000, 0x0001, 0x0002, 0x0004, 0x0005, 0x0006, 0x0007, 0x0008,
0x0009, 0x000A, 0x000B, 0x000C, 0x000D, 0x000E, 0x000F, 0x0010,
0x0011, 0x0012, 0x0013, 0x0014, 0x0015, 0x0016, 0x0017, 0x0018,
0x0019, 0x001A, 0x001B, 0x001C, 0x001D, 0x001E, 0x001F, 0x0020,
0x0021, 0x0022, 0x0023, 0x0024, 0x0025, 0x0026, 0x0027, 0x0028,
0x0029, 0x002A, 0x002B, 0x002C, 0x002D, 0x002E, 0x002F, 0x0030,
0x0031, 0x0032, 0x0033, 0x0034, 0x0035, 0x0036, 0x0037, 0x0038,
0x0039, 0x003A, 0x003B, 0x003C, 0x003D, 0x003E, 0x003F, 0x0040,
0x0041, 0x0042, 0x0043, 0x0043, 0x0044, 0x0045, 0x0046, 0x0047,
0x0048, 0x0049, 0x004A, 0x004B, 0x004C, 0x004D, 0x004E, 0x004F,
0x0050, 0x0051, 0x0052, 0x0053, 0x0054, 0x0054, 0x0055, 0x0056,
0x0057, 0x0058, 0x0059, 0x005A, 0x005B, 0x005C, 0x005D, 0x005E,
0x005F, 0x0060, 0x0060, 0x0061, 0x0062, 0x0063, 0x0064, 0x0065,
0x0066, 0x0067, 0x0068, 0x0069, 0x006A, 0x006B, 0x006B, 0x006C,
0x006D, 0x006E, 0x006F, 0x0070, 0x0071, 0x0072, 0x0073, 0x0074,
0x0075, 0x0075, 0x0076, 0x0077, 0x0078, 0x0079, 0x007A, 0x007B,
0x007C, 0x007D, 0x007E, 0x007E, 0x007F, 0x0080, 0x0081, 0x0082,
0x0083, 0x0084, 0x0085, 0x0086, 0x0086, 0x0087, 0x0088, 0x0089,
0x008A, 0x008B, 0x008C, 0x008D, 0x008E, 0x008E, 0x008F, 0x0090,
0x0091, 0x0092, 0x0093, 0x0094, 0x0095, 0x0096, 0x0096, 0x0097,
0x0098, 0x0099, 0x009A, 0x009B, 0x009C, 0x009D, 0x009E, 0x009E,
0x009F, 0x00A0, 0x00A1, 0x00A2, 0x00A3, 0x00A4, 0x00A5, 0x00A5,
0x00A6, 0x00A7, 0x00A8, 0x00A9, 0x00AA, 0x00AB, 0x00AC, 0x00AC,
0x00AD, 0x00AE, 0x00AF, 0x00B0, 0x00B1, 0x00B2, 0x00B3, 0x00B3,
0x00B4, 0x00B5, 0x00B6, 0x00B7, 0x00B8, 0x00B9, 0x00BA, 0x00BB,
0x00BB, 0x00BC, 0x00BD, 0x00BE, 0x00BF, 0x00C0, 0x00C1, 0x00C2,
0x00C2, 0x00C3, 0x00C4, 0x00C5, 0x00C6, 0x00C7, 0x00C8, 0x00C9,
0x00C9, 0x00CA, 0x00CB, 0x00CC, 0x00CD, 0x00CE, 0x00CF, 0x00D0,
0x00D0, 0x00D1, 0x00D2, 0x00D3, 0x00D4, 0x00D5, 0x00D6, 0x00D7,
0x00D7, 0x00D8, 0x00D9, 0x00DA, 0x00DB, 0x00DC, 0x00DD, 0x00DE,
0x00DF, 0x00DF, 0x00E0, 0x00E1, 0x00E2, 0x00E3, 0x00E4, 0x00E5,
0x00E6, 0x00E6, 0x00E7, 0x00E8, 0x00E9, 0x00EA, 0x00EB, 0x00EC,
0x00ED, 0x00EE, 0x00EE, 0x00EF, 0x00F0, 0x00F1, 0x00F2, 0x00F3,
0x00F4, 0x00F5, 0x00F6, 0x00F6, 0x00F7, 0x00F8, 0x00F9, 0x00FA,
0x00FB, 0x00FC, 0x00FD, 0x00FE, 0x00FE, 0x00FF, 0x0100, 0x0101,
0x0102, 0x0103, 0x0104, 0x0105, 0x0106, 0x0107, 0x0107, 0x0108,
0x0109, 0x010A, 0x010B, 0x010C, 0x010D, 0x010E, 0x010F, 0x0110,
0x0110, 0x0111, 0x0112, 0x0113, 0x0114, 0x0115, 0x0116, 0x0117,
0x0118, 0x0119, 0x011A, 0x011A, 0x011B, 0x011C, 0x011D, 0x011E,
0x011F, 0x0120, 0x0121, 0x0122, 0x0123, 0x0124, 0x0125, 0x0126,
0x0126, 0x0127, 0x0128, 0x0129, 0x012A, 0x012B, 0x012C, 0x012D,
0x012E, 0x012F, 0x0130, 0x0131, 0x0132, 0x0133, 0x0133, 0x0134,
0x0135, 0x0136, 0x0137, 0x0138, 0x0139, 0x013A, 0x013B, 0x013C,
0x013D, 0x013E, 0x013F, 0x0140, 0x0141, 0x0142, 0x0143, 0x0144,
0x0144, 0x0145, 0x0146, 0x0147, 0x0148, 0x0149, 0x014A, 0x014B,
0x014C, 0x014D, 0x014E, 0x014F, 0x0150, 0x0151, 0x0152, 0x0153,
0x0154, 0x0155, 0x0156, 0x0157, 0x0158, 0x0159, 0x015A, 0x015B,
0x015C, 0x015D, 0x015E, 0x015F, 0x0160, 0x0161, 0x0162, 0x0163,
0x0164, 0x0165, 0x0166, 0x0167, 0x0168, 0x0169, 0x016A, 0x016B,
0x016C, 0x016D, 0x016E, 0x016F, 0x0170, 0x0171, 0x0172, 0x0173,
0x0174, 0x0175, 0x0176, 0x0177, 0x0178, 0x0179, 0x017A, 0x017B,
```

```
   0x017C, 0x017D, 0x017E, 0x017F, 0x0180, 0x0181, 0x0182, 0x0183,
   0x0184, 0x0185, 0x0186, 0x0188, 0x0189, 0x018A, 0x018B, 0x018C,
   0x018D, 0x018E, 0x018F, 0x0190, 0x0191, 0x0192, 0x0193, 0x0194,
   0x0195, 0x0197, 0x0198, 0x0199, 0x019A, 0x019B, 0x019C, 0x019D,
   0x019E, 0x019F, 0x01A0, 0x01A1, 0x01A3, 0x01A4, 0x01A5, 0x01A6,
   0x01A7, 0x01A8, 0x01A9, 0x01AA, 0x01AC, 0x01AD, 0x01AE, 0x01AF,
   0x01B0, 0x01B1, 0x01B2, 0x01B3, 0x01B5, 0x01B6, 0x01B7, 0x01B8,
   0x01B9, 0x01BA, 0x01BC, 0x01BD, 0x01BE, 0x01BF, 0x01C0, 0x01C1,
   0x01C3, 0x01C4, 0x01C5, 0x01C6, 0x01C7, 0x01C9, 0x01CA, 0x01CB,
   0x01CC, 0x01CD, 0x01CF, 0x01D0, 0x01D1, 0x01D2, 0x01D3, 0x01D5,
   0x01D6, 0x01D7, 0x01D8, 0x01DA, 0x01DB, 0x01DC, 0x01DD, 0x01DF,
   0x01E0, 0x01E1, 0x01E2, 0x01E4, 0x01E5, 0x01E6, 0x01E7, 0x01E9,
   0x01EA, 0x01EB, 0x01ED, 0x01EE, 0x01EF, 0x01F1, 0x01F2, 0x01F3,
   0x01F5, 0x01F6, 0x01F7, 0x01F8, 0x01FA, 0x01FB, 0x01FC, 0x01FE,
   0x01FF, 0x0201, 0x0202, 0x0203, 0x0205, 0x0206, 0x0207, 0x0209,
   0x020A, 0x020C, 0x020D, 0x020E, 0x0210, 0x0211, 0x0213, 0x0214,
   0x0215, 0x0217, 0x0218, 0x021A, 0x021B, 0x021D, 0x021E, 0x0220,
   0x0221, 0x0223, 0x0224, 0x0225, 0x0227, 0x0228, 0x022A, 0x022B,
   0x022D, 0x022F, 0x0230, 0x0232, 0x0233, 0x0235, 0x0236, 0x0238,
   0x0239, 0x023B, 0x023C, 0x023E, 0x0240, 0x0241, 0x0243, 0x0244,
   0x0246, 0x0248, 0x0249, 0x024B, 0x024D, 0x024E, 0x0250, 0x0252,
   0x0253, 0x0255, 0x0257, 0x0258, 0x025A, 0x025C, 0x025D, 0x025F,
   0x0261, 0x0263, 0x0264, 0x0266, 0x0268, 0x026A, 0x026B, 0x026D,
   0x026F, 0x0271, 0x0273, 0x0275, 0x0276, 0x0278, 0x027A, 0x027C,
   0x027E, 0x0280, 0x0282, 0x0284, 0x0285, 0x0287, 0x0289, 0x028B,
   0x028D, 0x028F, 0x0291, 0x0293, 0x0295, 0x0297, 0x0299, 0x029B,
   0x029D, 0x029F, 0x02A1, 0x02A4, 0x02A6, 0x02A8, 0x02AA, 0x02AC,
   0x02AE, 0x02B0, 0x02B3, 0x02B5, 0x02B7, 0x02B9, 0x02BB, 0x02BE,
   0x02C0, 0x02C2, 0x02C5, 0x02C7, 0x02C9, 0x02CC, 0x02CE, 0x02D0,
   0x02D3, 0x02D5, 0x02D8, 0x02DA, 0x02DC, 0x02DF, 0x02E1, 0x02E4,
   0x02E7, 0x02E9, 0x02EC, 0x02EE, 0x02F1, 0x02F4, 0x02F6, 0x02F9,
   0x02FC, 0x02FE, 0x0301, 0x0304, 0x0307, 0x0309, 0x030C, 0x030F,
   0x0312, 0x0315, 0x0318, 0x031B, 0x031E, 0x0321, 0x0324, 0x0327,
   0x032A, 0x032D, 0x0330, 0x0334, 0x0337, 0x033A, 0x033D, 0x0341,
   0x0344, 0x0348, 0x034B, 0x034F, 0x0352, 0x0356, 0x0359, 0x035D,
   0x0360, 0x0364, 0x0368, 0x036C, 0x0370, 0x0373, 0x0377, 0x037B,
   0x037F, 0x0383, 0x0388, 0x038C, 0x0390, 0x0394, 0x0399, 0x039D,
   0x03A2, 0x03A6, 0x03AB, 0x03AF, 0x03B4, 0x03B9, 0x03BE, 0x03C3,
   0x03C8, 0x03CD, 0x03D2, 0x03D7, 0x03DD, 0x03E2, 0x03E8, 0x03ED,
   0x03F3, 0x03F9, 0x03FF, 0x0405, 0x040B, 0x0411, 0x0418, 0x041E,
   0x0425, 0x042C, 0x0433, 0x043A, 0x0441, 0x0448, 0x0450, 0x0458,
   0x0460, 0x0468, 0x0470, 0x0479, 0x0482, 0x048B, 0x0494, 0x049D,
   0x04A7, 0x04B1, 0x04BC, 0x04C6, 0x04D1, 0x04DD, 0x04E9, 0x04F5,
   0x0501, 0x050F, 0x051C, 0x052B, 0x0539, 0x0549, 0x0559, 0x056A,
   0x057C, 0x058E, 0x05A2, 0x05B7, 0x05CD, 0x05E5, 0x05FE, 0x0619,
   0x0636, 0x0655, 0x0677, 0x069D, 0x06C6, 0x06F4, 0x0728, 0x0763,
   0x07A8, 0x07FA, 0x085E, 0x08DE, 0x098C, 0x0A96, 0x0C9D, 0x7FFF};

int_fast16_t adc2temp(int_fast16_t temp)
{
   if (temp < 1024 & temp >= 0)
   {
      return pgm_read_word(tempcnv + temp);
   }
   else
   {
      return 0xF555;
   }

}
```

A.3.2 H-Bridge & Servo Control PWM Driver Libraries

pwm.h

```
/*
 * pwm.h
 *
 * Created: 11/21/2013 6:21:48 PM
 *  Author: Nick
 */


#ifndef PWM_H_
#define PWM_H_

/*
Temperature & Motion Control Peripherals

H-Bridges:

   TEC:
   Heat: PD4
   Cool: PD5
   PWM:  PC4/OC3C

   Pump1:
   In:      PA6
   Out:  PA7
   PWM:  PC5/OC3B

   Pump2:
   In:      PA4
   Out:  PA5
   PWM:  PC6/OC3A

   Pump3:
   In:      PA2
   Out:  PA3
   PWM:  PB6/OC1B

   Pump4:
   In:      PA0
   Out:  PA1
   PWM:  PB5/OC1A

Servos:

   PWM:  PB4/OC2A

   Enable:  PC3

   SEL0: PC0
   SEL1: PC1
   SEL2: PC2

*/

#include <stdint.h>

#define tec_dir_port PORTD
#define tec_cool_pin PD4
#define tec_heat_pin PD5

#define pump_dir_port   PORTA
#define pump1_in_pin PA6
#define pump1_out_pin   PA7
#define pump2_in_pin PA4
#define pump2_out_pin   PA5
```

```c
#define pump3_in_pin PA2
#define pump3_out_pin   PA3
#define pump4_in_pin PA0
#define pump4_out_pin   PA1

#define servo_sel_port  PORTC
#define servo_sel_mask  0x07
#define servo_en_pin PC3
#define servo_s0_pin PC0
#define servo_s1_pin PC1
#define servo_s2_pin PC2

#define tec_pw_reg      OCR3C
#define pump1_pw_reg OCR3B
#define pump2_pw_reg OCR3A
#define pump3_pw_reg OCR1B
#define pump4_pw_reg OCR1A
#define servo_pw_reg OCR2A

enum tec_mode  {TEC_OFF, TEC_COOL, TEC_HEAT};
enum pump_mode {MODE_NULL, MODE_OFF, MODE_IN, MODE_OUT};
enum pump      {PUMP_NULL, PUMP1, PUMP2, PUMP3, PUMP4};
enum servo     {SERVO1, SERVO2, SERVO3, SERVO4, SERVO5, SERVO6, SERVO7, SERVO8};

struct pump_status {
   enum pump_mode mode;
   int_fast16_t pw;
   };

struct tec_status {
   enum tec_mode mode;
   uint_fast16_t pw;
   };

void configure_pwm(void);

void set_tec(enum tec_mode mode, uint_fast16_t pw);

struct tec_status get_tec();

void set_servo(enum servo s, uint8_t pw);

void set_pump(enum pump p, enum pump_mode mode, uint_fast16_t pw);

struct pump_status get_pump(enum pump p);

void scram();

#endif /* PWM_H_ */
```

pwm.c

```c
/*
Temperature & Motion Control Peripherals

H-Bridges:

   TEC:
   Heat: PD4
   Cool: PD5
   PWM:  PC4/OC3C

   Pump1:
   In:      PA6
```

```
   Out:  PA7
   PWM:  PC5/OC3B

   Pump2:
   In:     PA4
   Out:  PA5
   PWM:  PC6/OC3A

   Pump3:
   In:     PA2
   Out:  PA3
   PWM:  PB6/OC1B

   Pump4:
   In:     PA0
   Out:  PA1
   PWM:  PB5/OC1A

Servos:

   PWM:  PB4/OC2A

   Enable:  PC3

   SEL0: PC0
   SEL1: PC1
   SEL2: PC2

*/

#include "AntennaController_v2_1.h"
#include "pwm.h"
#include <avr/io.h>
#include <util/delay.h>

void configure_pwm(void) {
   // Configure Pump and TEC PWM Direction & PWM Outputs:

   DDRA |=  0b11111111; //PA0:7 Outputs: Pump1-4 Direction Pins
   DDRB |= 0b01110000;  //PB4:6 Outputs: Servo & Pump 3&4 PWM Outputs
   DDRC |= 0b01111111;  //PC0:6 Outputs: Servo Select, Enable, TEC & Pump 1&2 PWM Outputs
   DDRD |= 0b00110000;  //PD4:5  Outputs: TEC Heat/Cool Pins

   OCR1A = 0x0;         //Set all PWM Outputs to 0
   OCR1B = 0x0;

   OCR2A = 0x0;

   OCR3A = 0x0;
   OCR3B = 0x0;
   OCR3C = 0x0;

   TCCR1A = 0b10101000; // Phase & Frequency Correct PWM, 16MHz Clock
   TCCR1B = 0b00010001;
   TCCR3A = 0b10101000;
   TCCR3B = 0b00010001;

   TCCR2A = 0b10000001; // Phase Correct PWM
   TCCR2B = 0b00000110; // Clock = 16MHz/256 -> f_PWM=16MHz/(256*510) = 122.55Hz

   ICR1 = 0x03FF; // TOP = 1023 -> f_PWM = 16e6/(2*1*1023) = 7.8201kHz
   ICR3 = 0x03FF;

   return;
```

```
}

void set_tec(enum tec_mode mode, uint_fast16_t pw) {
   // Limit pulse width to 10 bits
   uint_fast16_t pw_lim = 0;
   if (pw > 0x3FF)   {pw_lim = 0x3FF;}
   else        {pw_lim = pw;}

   switch (mode) {
      case TEC_OFF:  // TEC Off
         tec_dir_port &= ~((1<<tec_heat_pin)|(1<<tec_cool_pin));
         break;
      case TEC_COOL: // TEC Cool
         tec_dir_port |= (1<<tec_cool_pin);
         tec_dir_port &= ~(1<<tec_heat_pin);
         break;
      case TEC_HEAT: // TEC Heat
         tec_dir_port |= (1<<tec_heat_pin);
         tec_dir_port &= ~(1<<tec_cool_pin);
         break;
      default:
         return;
         break;
   }
   tec_pw_reg = pw_lim; // Set TEC PWM register
   return;
}

struct tec_status get_tec() {
   struct tec_status myTECstatus;

   if ( !(tec_dir_port&(1<<tec_heat_pin)) && !(tec_dir_port&(1<<tec_cool_pin)) ) {
      myTECstatus.mode = TEC_OFF;
   }
   if ( !(tec_dir_port&(1<<tec_heat_pin)) && (tec_dir_port&(1<<tec_cool_pin)) ) {
      myTECstatus.mode = TEC_COOL;
   }
   if ( (tec_dir_port*(1<<tec_heat_pin)) && !(tec_dir_port&(1<<tec_cool_pin)) ) {
      myTECstatus.mode = TEC_HEAT;
   }

   myTECstatus.pw = tec_pw_reg;

   return myTECstatus;
}

void set_servo(enum servo s, uint8_t pw) {
   if ((s < SERVO1)||(s > SERVO8))  { // Validate servo selection
      return;
   }

   servo_pw_reg = pw; // Set desired pulse width

   servo_sel_port = ((servo_sel_port & ~servo_sel_mask) | (s & 0x07));  // Set servo select pins

   servo_sel_port |= (1<<servo_en_pin);   // Enable servo output
   _delay_ms(SERVO_TIME);              // Wait for servo move to complete
   servo_sel_port &= ~(1<<servo_en_pin);  // Disable servo output

   servo_pw_reg = 0; // Reset pulse width to 0
}

void set_pump(enum pump p, enum pump_mode mode, uint_fast16_t pw) {
   // Limit pulse width to 10 bits
```

```
    uint_fast16_t pw_lim = 0;
    if (pw > 0x3FF)   {pw_lim = 0x3FF;}
    else        {pw_lim = pw;}

    switch (mode) {   // Set Pump Mode
       case MODE_OFF: // Pump Off
          pump_dir_port &= ~( (1 << (9-2*p)) | (1 << (8-2*p)) );
          break;
       case MODE_IN:  // Pump In
          pump_dir_port |=  (1 << (8-2*p));
          pump_dir_port &= ~(1 << (9-2*p));
          break;
       case MODE_OUT: // Pump Out
          pump_dir_port |=  (1 << (9-2*p));
          pump_dir_port &= ~(1 << (8-2*p));
          break;
       case MODE_NULL:
          break;
       default:
          return;
          break;
    }
    switch (p) {   // Set Pump pulse width
       case PUMP1:
          pump1_pw_reg = pw_lim;
          break;
       case PUMP2:
          pump2_pw_reg = pw_lim;
          break;
       case PUMP3:
          pump3_pw_reg = pw_lim;
          break;
       case PUMP4:
          pump4_pw_reg = pw_lim;
          break;
       case PUMP_NULL:
          break;
       default:
          return;
          break;
    }
}

struct pump_status get_pump(enum pump p) {
    struct pump_status myPumpStatus;
    switch (p) {
       case PUMP1:
          myPumpStatus.pw = pump1_pw_reg;
          break;
       case PUMP2:
          myPumpStatus.pw = pump2_pw_reg;
          break;
       case PUMP3:
          myPumpStatus.pw = pump3_pw_reg;
          break;
       case PUMP4:
          myPumpStatus.pw = pump4_pw_reg;
          break;
       default:
          myPumpStatus.pw = 0;
          break;
    }

    if ( !(pump_dir_port&(1<<(8-2*p))) && !(pump_dir_port*(1<<(9-2*p))) ) {
```

```
            myPumpStatus.mode = MODE_OFF;
        }
        if ( (pump_dir_port&(1<<(8-2*p))) && !(pump_dir_port&(1<<(9-2*p))) ) {
            myPumpStatus.mode = MODE_IN;
        }
        if ( !(pump_dir_port&(1<<(8-2*p))) && (pump_dir_port&(1<<(9-2*p))) ) {
            myPumpStatus.mode = MODE_OUT;
        }

        return myPumpStatus;
}

// Shut down all power outputs
void scram() {
    globalFlags.in_scram = 1;
    pump_dir_port = 0x0;
    tec_dir_port &= ~(0b00110000);
    servo_sel_port &= ~(0b00001111);
    return;
}
```

A.3.3 Phase Shifter Control DAC Driver Library

AD5668.h

```
/*
 * AD5668.h
 *
 * Created: 5/28/2014 11:01:38 AM
 *  Author: Nick
 */


#ifndef AD5668_H_
#define AD5668_H_

#include <stdint.h>

#define WR_IN        0b0000   // Write to input register n
#define UP_DAC       0b0001   // Update DAC register from input register n
#define WR_IN_UP_ALL 0b0010   // Write to input register n, update all DAC registers
#define WR_UP_DAC    0b0011   // Write & update DAC channel n
#define PU_PD        0b0100   // Toggle DAC powerup/powerdown
#define LD_CLR       0b0101   // Load clear code register
#define LD_LDAC      0b0110   // Load ~LDAC register
#define RST          0b0111   // Reset DAC
#define SET_REF      0b1000   // Configure INT/EXT reference voltage

enum dac_add {DAC_A=0x0, DAC_B=0x1, DAC_C=0x2, DAC_D=0x3,
              DAC_E=0x4, DAC_F=0x5, DAC_G=0x6, DAC_H=0x7, DAC_ALL=0xF};

void configure_dac();

void set_dac_word(enum dac_add dac, uint_fast16_t word);

#endif /* AD5668_H_ */
```

AD5668.c

```
/*
 * AD5668.c
 *
 * Created: 5/28/2014 11:01:27 AM
```

```c
 *  Author: Nick
 */

#include "AD5668.h"
#include "spi.h"

void configure_dac() {
   uint8_t cmd[4] = {0,0,0,0};
   //uint_fast32_t cmd = 0;

   cmd[0] = (SET_REF << 0);   // Setup reference
   cmd[3] = (1<<0);          // Internal reference on

   //cmd = (SET_REF << 24) | (1<<0);

   spi_transfer(cmd,4);
}

void set_dac_word(enum dac_add dac, uint_fast16_t word) {
   uint8_t cmd[4] = {0,0,0,0};
   //uint_fast32_t cmd = 0;

   cmd[0] = (WR_UP_DAC << 0); // Write & update DAC channel
   cmd[1] = (dac << 4);    // DAC address
   cmd[1] |= ((word & 0xF000) >> 12);
   cmd[2] = ((word & 0x0FF0) >> 4);
   cmd[3] = (((word & 0x000F) >> 0) << 4);

   //cmd = (WR_UP_DAC << 24) | (dac << 20) | (word << 4);

   spi_transfer(cmd,4);

}
```

A.4 Communication Code

A.4.1 XBee WiFi Radio Initialization Code

xbee.h

```c
/*
 * xbee.h
 *
 * Created: 11/21/2013 5:48:56 PM
 *  Author: Nick
 */


#ifndef XBEE_H_
#define XBEE_H_

void configure_xbee(void);



#endif /* XBEE_H_ */
```

xbee.c

```c
/*
Configures an XBee Wi-Fi in AT mode to connect to a Wi-Fi network
and alert the server that the device is ready.
*/
```

```c

#include <avr/pgmspace.h>
#include <util/delay.h>
#include <string.h>
#include "AntennaController_v2_1.h"
#include "serial.h"
#include "uart.h"

#define LED_ON (PORTD |= (1<<6))
#define LED_OFF   (PORTD &= ~(1<<6))

/**************************************************************************/
/* Control Server & Network Parameters                                    */
/**************************************************************************/

#define ssid   "aperskin-control"   // WiFi network SSID
#define pass   "aperphore"       // Encryption Passphrase

#define dest   "10.0.0.10"    // Control Server IP address

#define dport  "238C"            // Control Server Listening Port:   9100 = 0x238C
#define sport  "238D"            // XBee Listening Port:          9101 = 0x238D

void check_ok(void);

void configure_xbee()
{
   flush_serial();
   send_str_P(PSTR("+++"));      // Enter AT command mode
   _delay_ms(1100);     // Observe default 1000ms guard time around +++
   check_ok();

   send_str_P(PSTR("ATRE\r"));      // Reset to factory defaults
   _delay_ms(20);
   check_ok();

   send_str_P(PSTR("ATDL "));    // Set destination address
   send_str_P(PSTR(dest));
   send_str_P(PSTR("\r"));
   _delay_ms(20);
   check_ok();

   send_str_P(PSTR("ATC0 "));    // Set source (listening) port
   send_str_P(PSTR(sport));
   send_str_P(PSTR("\r"));
   _delay_ms(20);
   check_ok();

   send_str_P(PSTR("ATDE "));    // Set destination port
   send_str_P(PSTR(dport));
   send_str_P(PSTR("\r"));
   _delay_ms(20);
   check_ok();

   send_str_P(PSTR("ATID "));    // Set WiFi SSID
   send_str_P(PSTR(ssid));
   send_str_P(PSTR("\r"));
   _delay_ms(50);
   check_ok();

   send_str_P(PSTR("ATPK "));    // Set encryption passphrase
   send_str_P(PSTR(pass));
   send_str_P(PSTR("\r"));
   _delay_ms(50);
```

```c
   check_ok();

   send_str_P(PSTR("ATEE 2\r")); // Enable WPA2 encryption
   _delay_ms(20);
   check_ok();

   send_str_P(PSTR("ATIP 1\r")); // Set TCP mode
   _delay_ms(20);
   check_ok();

   send_str_P(PSTR("ATAC\r"));      // Apply configuration changes
   _delay_ms(20);
   check_ok();

/* END OF CONFIGURATION COMMANDS */

   flush_serial();

   char sh_buf[10], sl_buf[10];

   send_str_P((PSTR("ATSH\r"))); // Get high 4 bytes of XBee serial
   _delay_ms(20);

   recv_str(sh_buf,10);
   strncat(deviceSerial, sh_buf, 10);

   send_str_P(PSTR("ATSL\r"));      // Get low 4 bytes of XBee serial
   _delay_ms(20);

   recv_str(sl_buf,10);
   strncat(deviceSerial, sl_buf, 10);

   flush_serial();

   /*
   Poll XBee connection status until it returns 0x0
   0x00: AP join successful, IP address assigned, listening socket established
   */
   char con_reply[10] = "\0";
   while (1)
   {
      LED_ON;
      send_str_P(PSTR("ATAI\r"));
      _delay_ms(200);
      recv_str(con_reply,10);
      LED_OFF;
      _delay_ms(200);
      if (strcasecmp_P(con_reply, PSTR("0"))==0)
      {
         break;
      }
   }

   send_str_P(PSTR("ATCN\r"));      // Exit command mode
   _delay_ms(20);
   check_ok();

   _delay_ms(500);

   flush_serial();

   send_str(deviceSerial);
   send_str_P(PSTR(" READY"));      // Send READY string to server
   return;
```

```
}

// Check if XBee replied "OK" to a command, Blink "SOS" if not
void check_ok(void)
{
   char buf[10];
   recv_str(buf,10);
   if (strcasecmp_P(buf,PSTR("OK"))!=0)
   {
      while (1) // Blink "SOS"
      {
         LED_OFF;
         _delay_ms(300);

         // Blink "S"
         LED_ON;
         _delay_ms(75);
         LED_OFF;
         _delay_ms(75);
         LED_ON;
         _delay_ms(75);
         LED_OFF;
         _delay_ms(75);
         LED_ON;
         _delay_ms(75);
         LED_OFF;
         _delay_ms(225);

         // Blink "O"
         LED_ON;
         _delay_ms(225);
         LED_OFF;
         _delay_ms(75);
         LED_ON;
         _delay_ms(225);
         LED_OFF;
         _delay_ms(75);
         LED_ON;
         _delay_ms(225);
         LED_OFF;
         _delay_ms(225);

         // Blink "S"
         LED_ON;
         _delay_ms(75);
         LED_OFF;
         _delay_ms(75);
         LED_ON;
         _delay_ms(75);
         LED_OFF;
         _delay_ms(75);
         LED_ON;
         _delay_ms(75);
         LED_OFF;
         _delay_ms(225);
      }
   }
   else return;
}
```

A.4.2 Serial String Communication Library

serial.h

```c
#ifndef SERIAL_H_
#define SERIAL_H_


#include <stdint.h>

void flush_serial();

void send_str_P(const char *s);

void send_str(char *s);

uint8_t recv_str(char *buf, uint8_t size);

#endif /* SERIAL_H_ */
```

serial.c

```c
/*
Functions for sending and receiving strings via UART serial port
*/

#include <avr/pgmspace.h>

#include "AntennaController_v2_1.h"
#include "serial.h"
#include "usb_serial.h"
#include "uart.h"

// Flush buffered command responses
void flush_serial() {
   #ifdef USB_SERIAL
   usb_serial_flush_input();
   #else
   while (uart_available()) {
      uart_getchar();
   }
   #endif
}

// Send string in flash memory (program space) to serial port
void send_str_P(const char *s) {
   char c;
   while (1) {
      c = pgm_read_byte(s++);
      if (!c) break;
#ifdef USB_SERIAL
      usb_serial_putchar(c);
#else
      uart_putchar(c);
#endif
   }
}

// Send a string in RAM to serial port
void send_str(char *s) {
   char c;
   for (int i = 0; i < 255; i++) {
      c = s[i];
      if (!c) break;
#ifdef USB_SERIAL
      usb_serial_putchar(c);
#else
```

```c
        uart_putchar(c);
#endif

    }
}

// Receive a \r terminated string from serial port into a buffer
uint8_t recv_str(char *buf, uint8_t size) {
    char inChar;
    uint8_t count=0;
#ifdef USB_SERIAL
    while ((usb_serial_available() > 0) && (count <= size)) {
        inChar = usb_serial_getchar();
#else
    while ((uart_available() > 0) && (count <= size)) {
        inChar = uart_getchar();
#endif
        if (inChar >= 0x20 && inChar <= 0x7E) {   // Add to buffer if printable
            *buf++ = inChar;
            *buf = '\0';
            count++;
        }
        if (inChar == '\r')  { // Return buffer if carriage return received
            *buf = '\0';
            return count;
        }
    }
    *buf = '\0';
    return count;
}
```

A.4.3 SPI Master Controller Driver Library

spi.h

```c
/*
 * spi.h
 *
 * Created: 5/28/2014 12:41:29 AM
 *  Author: Nick
 */


#ifndef SPI_H_
#define SPI_H_

#define spi_port     PORTB
#define spi_port_dir DDRB

#define SS     PB0
#define SCLK   PB1
#define MOSI   PB2
#define MISO   PB3

void configure_spi();

uint8_t spi_transfer(char* buf, uint8_t length);

#endif /* SPI_H_ */
```

spi.c

```c
/*
 * spi.c
```

```
 *
 * Created: 5/28/2014 12:41:17 AM
 *  Author: Nick
 */

/*
SPI Master driver for AT90USB1286
Pin assignments:

SS:      PB0
SCLK: PB1
MOSI: PB2
MISO: PB3
*/
#include <avr/io.h>

#include "spi.h"

void configure_spi() {
   spi_port_dir |= ((1<<MOSI)|(1<<SCLK)|(1<<SS));  // MOSI, SCLK, SS outputs
   spi_port_dir &= ~(1<<MISO);                     // MISO input

   SPCR = (0<<SPIE)|    // SPI Interrupt off
         (1<<SPE)|      // SPI Enabled
         (0<DORD)|      // MSB first
         (1<<MSTR)|     // SPI Master mode
         (0<<CPOL)|     // SCK low idle
         (1<<CPHA)|     // Setup on leading edge, Sample on trailing edge
         (0b01<<SPR0);  // fSCK = 16MHz/4 = 4 MHz

   spi_port |= (1<<SS); // Set ~SS pin high to disable comm to start
}

// Shift character array out via SPI, store received data back in character array
uint8_t spi_transfer(char* buf, uint8_t length) {
      spi_port &= ~(1<<SS);    // Pull ~SS low to select slave

      uint8_t status, inData;

      for (uint8_t i = 0; i < length; i++) {
         SPDR = buf[i]; // Transmit character from buffer
         // Wait for interrpt flag to signal sucessful transmission
         while (!(SPSR & (1<<SPIF))) ;
         status = SPSR; // Read status register (clears SPIF when set)
         inData = SPDR; // Grab incoming data byte

         //if (!(SPCR & (1<<MSTR))) {return 1;} // SPI Master bit no longer set
         //if (status & (1<<WCOL)) {return 2;}     // SPI write collision occurred

         buf[i] = inData;  // Store received data byte back in buffer
      }

      spi_port |= (1<<SS); // Bring ~SS high to deselect slave
      return 0;
}
```

A.4.4 UART & USB Serial Port Driver Libraries

uart.h

```
#ifndef UART_H_
#define UART_H_

#include <stdint.h>
```

```c

void uart_init(uint32_t baud);
void uart_putchar(uint8_t c);
uint8_t uart_getchar(void);
uint8_t uart_available(void);

#endif /* UART_H_ */
```

uart.c

```c
/* UART Example for Teensy USB Development Board
 * http://www.pjrc.com/teensy/
 * Copyright (c) 2009 PJRC.COM, LLC
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 * THE SOFTWARE.
 */

// Version 1.0: Initial Release
// Version 1.1: Add support for Teensy 2.0, minor optimizations


#include <avr/io.h>
#include <avr/interrupt.h>

#include "uart.h"

// These buffers may be any size from 2 to 256 bytes.
#define RX_BUFFER_SIZE 64
#define TX_BUFFER_SIZE 40

static volatile uint8_t tx_buffer[TX_BUFFER_SIZE];
static volatile uint8_t tx_buffer_head;
static volatile uint8_t tx_buffer_tail;
static volatile uint8_t rx_buffer[RX_BUFFER_SIZE];
static volatile uint8_t rx_buffer_head;
static volatile uint8_t rx_buffer_tail;

// Initialize the UART
void uart_init(uint32_t baud)
{
   cli();
   UBRR1 = (F_CPU / 4 / baud - 1) / 2;
   UCSR1A = (1<<U2X1);
   UCSR1B = (1<<RXEN1) | (1<<TXEN1) | (1<<RXCIE1);
   UCSR1C = (1<<UCSZ11) | (1<<UCSZ10);
   tx_buffer_head = tx_buffer_tail = 0;
   rx_buffer_head = rx_buffer_tail = 0;
   sei();
```

```
}

// Transmit a byte
void uart_putchar(uint8_t c)
{
   uint8_t i;

   i = tx_buffer_head + 1;
   if (i >= TX_BUFFER_SIZE) i = 0;
   while (tx_buffer_tail == i) ; // wait until space in buffer
   cli();   //commented out in original UART source
   tx_buffer[i] = c;
   tx_buffer_head = i;
   UCSR1B = (1<<RXEN1) | (1<<TXEN1) | (1<<RXCIE1) | (1<<UDRIE1);
   sei();   //commented out in original UART source
}

// Receive a byte
uint8_t uart_getchar(void)
{
        uint8_t c, i;

   while (rx_buffer_head == rx_buffer_tail) ; // wait for character
        i = rx_buffer_tail + 1;
        if (i >= RX_BUFFER_SIZE) i = 0;
        c = rx_buffer[i];
        rx_buffer_tail = i;
        return c;
}

// Return the number of bytes waiting in the receive buffer.
// Call this before uart_getchar() to check if it will need
// to wait for a byte to arrive.
uint8_t uart_available(void)
{
   uint8_t head, tail;

   head = rx_buffer_head;
   tail = rx_buffer_tail;
   if (head >= tail) return head - tail;
   return RX_BUFFER_SIZE + head - tail;
}

// Transmit Interrupt
ISR(USART1_UDRE_vect)
{
   uint8_t i;

   if (tx_buffer_head == tx_buffer_tail) {
      // buffer is empty, disable transmit interrupt
      UCSR1B = (1<<RXEN1) | (1<<TXEN1) | (1<<RXCIE1);
   } else {
      i = tx_buffer_tail + 1;
      if (i >= TX_BUFFER_SIZE) i = 0;
      UDR1 = tx_buffer[i];
      tx_buffer_tail = i;
   }
}

// Receive Interrupt
ISR(USART1_RX_vect)
{
   uint8_t c, i;
```

```
    c = UDR1;
    i = rx_buffer_head + 1;
    if (i >= RX_BUFFER_SIZE) i = 0;
    if (i != rx_buffer_tail) {
       rx_buffer[i] = c;
       rx_buffer_head = i;
    }
}
```

usb_serial.h

```
#ifndef usb_serial_h__
#define usb_serial_h__

#include <stdint.h>

// setup
void usb_init(void);          // initialize everything
uint8_t usb_configured(void);      // is the USB port configured

// receiving data
int16_t usb_serial_getchar(void);   // receive a character (-1 if timeout/error)
uint8_t usb_serial_available(void);   // number of bytes in receive buffer
void usb_serial_flush_input(void);   // discard any buffered input

// transmitting data
int8_t usb_serial_putchar(uint8_t c);   // transmit a character
int8_t usb_serial_putchar_nowait(uint8_t c);  // transmit a character, do not wait
int8_t usb_serial_write(const uint8_t *buffer, uint16_t size); // transmit a buffer
void usb_serial_flush_output(void);   // immediately transmit any buffered output

// serial parameters
uint32_t usb_serial_get_baud(void);   // get the baud rate
uint8_t usb_serial_get_stopbits(void);   // get the number of stop bits
uint8_t usb_serial_get_paritytype(void);// get the parity type
uint8_t usb_serial_get_numbits(void);   // get the number of data bits
uint8_t usb_serial_get_control(void);   // get the RTS and DTR signal state
int8_t usb_serial_set_control(uint8_t signals); // set DSR, DCD, RI, etc

// constants corresponding to the various serial parameters
#define USB_SERIAL_DTR         0x01
#define USB_SERIAL_RTS         0x02
#define USB_SERIAL_1_STOP      0
#define USB_SERIAL_1_5_STOP      1
#define USB_SERIAL_2_STOP      2
#define USB_SERIAL_PARITY_NONE      0
#define USB_SERIAL_PARITY_ODD      1
#define USB_SERIAL_PARITY_EVEN      2
#define USB_SERIAL_PARITY_MARK      3
#define USB_SERIAL_PARITY_SPACE      4
#define USB_SERIAL_DCD         0x01
#define USB_SERIAL_DSR         0x02
#define USB_SERIAL_BREAK      0x04
#define USB_SERIAL_RI         0x08
#define USB_SERIAL_FRAME_ERR      0x10
#define USB_SERIAL_PARITY_ERR      0x20
#define USB_SERIAL_OVERRUN_ERR      0x40

// This file does not include the HID debug functions, so these empty
// macros replace them with nothing, so users can compile code that
// has calls to these functions.
#define usb_debug_putchar(c)
#define usb_debug_flush_output()
```

```c
// Everything below this point is only intended for usb_serial.c
#ifdef USB_SERIAL_PRIVATE_INCLUDE
#include <avr/io.h>
#include <avr/pgmspace.h>
#include <avr/interrupt.h>

#define EP_TYPE_CONTROL          0x00
#define EP_TYPE_BULK_IN          0x81
#define EP_TYPE_BULK_OUT         0x80
#define EP_TYPE_INTERRUPT_IN     0xC1
#define EP_TYPE_INTERRUPT_OUT    0xC0
#define EP_TYPE_ISOCHRONOUS_IN   0x41
#define EP_TYPE_ISOCHRONOUS_OUT  0x40
#define EP_SINGLE_BUFFER         0x02
#define EP_DOUBLE_BUFFER         0x06
#define EP_SIZE(s)   ((s) == 64 ? 0x30 :   \
         ((s) == 32 ? 0x20 :   \
         ((s) == 16 ? 0x10 :   \
                      0x00)))

#define MAX_ENDPOINT      4

#define LSB(n) (n & 255)
#define MSB(n) ((n >> 8) & 255)

#if defined(__AVR_AT90USB162__)
#define HW_CONFIG()
#define PLL_CONFIG() (PLLCSR = ((1<<PLLE)|(1<<PLLP0)))
#define USB_CONFIG() (USBCON = (1<<USBE))
#define USB_FREEZE() (USBCON = ((1<<USBE)|(1<<FRZCLK)))
#elif defined(__AVR_ATmega32U4__)
#define HW_CONFIG() (UHWCON = 0x01)
#define PLL_CONFIG() (PLLCSR = 0x12)
#define USB_CONFIG() (USBCON = ((1<<USBE)|(1<<OTGPADE)))
#define USB_FREEZE() (USBCON = ((1<<USBE)|(1<<FRZCLK)))
#elif defined(__AVR_AT90USB646__)
#define HW_CONFIG() (UHWCON = 0x81)
#define PLL_CONFIG() (PLLCSR = 0x1A)
#define USB_CONFIG() (USBCON = ((1<<USBE)|(1<<OTGPADE)))
#define USB_FREEZE() (USBCON = ((1<<USBE)|(1<<FRZCLK)))
#elif defined(__AVR_AT90USB1286__)
#define HW_CONFIG() (UHWCON = 0x81)
#define PLL_CONFIG() (PLLCSR = 0x16)
#define USB_CONFIG() (USBCON = ((1<<USBE)|(1<<OTGPADE)))
#define USB_FREEZE() (USBCON = ((1<<USBE)|(1<<FRZCLK)))
#endif

// standard control endpoint request types
#define GET_STATUS          0
#define CLEAR_FEATURE          1
#define SET_FEATURE          3
#define SET_ADDRESS          5
#define GET_DESCRIPTOR          6
#define GET_CONFIGURATION       8
#define SET_CONFIGURATION       9
#define GET_INTERFACE          10
#define SET_INTERFACE          11
// HID (human interface device)
#define HID_GET_REPORT          1
#define HID_GET_PROTOCOL       3
#define HID_SET_REPORT          9
#define HID_SET_IDLE          10
```

```
#define HID_SET_PROTOCOL      11
// CDC (communication class device)
#define CDC_SET_LINE_CODING      0x20
#define CDC_GET_LINE_CODING      0x21
#define CDC_SET_CONTROL_LINE_STATE   0x22
#endif
#endif
```

usb_serial.c

```
/* USB Serial Example for Teensy USB Development Board
 * http://www.pjrc.com/teensy/usb_serial.html
 * Copyright (c) 2008,2010,2011 PJRC.COM, LLC
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
 * AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
 * LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
 * OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN
 * THE SOFTWARE.
 */

// Version 1.0: Initial Release
// Version 1.1: support Teensy++
// Version 1.2: fixed usb_serial_available
// Version 1.3: added transmit bandwidth test
// Version 1.4: added usb_serial_write
// Version 1.5: add support for Teensy 2.0
// Version 1.6: fix zero length packet bug
// Version 1.7: fix usb_serial_set_control

#define USB_SERIAL_PRIVATE_INCLUDE
#include "usb_serial.h"


/**************************************************************************
 *
 *  Configurable Options
 *
 **************************************************************************/

// You can change these to give your code its own name.  On Windows,
// these are only used before an INF file (driver install) is loaded.
#define STR_MANUFACTURER   L"Your Name"
#define STR_PRODUCT      L"USB Serial"

// All USB serial devices are supposed to have a serial number
// (according to Microsoft).  On windows, a new COM port is created
// for every unique serial/vendor/product number combination.  If
// you program 2 identical boards with 2 different serial numbers
// and they are assigned COM7 and COM8, each will always get the
// same COM port number because Windows remembers serial numbers.
//
```

```c
// On Mac OS-X, a device file is created automatically which
// incorperates the serial number, eg, /dev/cu-usbmodem12341
//
// Linux by default ignores the serial number, and creates device
// files named /dev/ttyACM0, /dev/ttyACM1... in the order connected.
// Udev rules (in /etc/udev/rules.d) can define persistent device
// names linked to this serial number, as well as permissions, owner
// and group settings.
#define STR_SERIAL_NUMBER  L"12345"

// Mac OS-X and Linux automatically load the correct drivers.  On
// Windows, even though the driver is supplied by Microsoft, an
// INF file is needed to load the driver.  These numbers need to
// match the INF file.
#define VENDOR_ID    0x16C0
#define PRODUCT_ID      0x047A

// When you write data, it goes into a USB endpoint buffer, which
// is transmitted to the PC when it becomes full, or after a timeout
// with no more writes.  Even if you write in exactly packet-size
// increments, this timeout is used to send a "zero length packet"
// that tells the PC no more data is expected and it should pass
// any buffered data to the application that may be waiting.  If
// you want data sent immediately, call usb_serial_flush_output().
#define TRANSMIT_FLUSH_TIMEOUT   5   /* in milliseconds */


// If the PC is connected but not "listening", this is the length
// of time before usb_serial_getchar() returns with an error.  This
// is roughly equivilant to a real UART simply transmitting the
// bits on a wire where nobody is listening, except you get an error
// code which you can ignore for serial-like discard of data, or
// use to know your data wasn't sent.
#define TRANSMIT_TIMEOUT   25   /* in milliseconds */

// USB devices are supposed to implment a halt feature, which is
// rarely (if ever) used.  If you comment this line out, the halt
// code will be removed, saving 116 bytes of space (gcc 4.3.0).
// This is not strictly USB compliant, but works with all major
// operating systems.
#define SUPPORT_ENDPOINT_HALT



/**************************************************************************
 *
 *  Endpoint Buffer Configuration
 *
 **************************************************************************/

// These buffer sizes are best for most applications, but perhaps if you
// want more buffering on some endpoint at the expense of others, this
// is where you can make such changes.  The AT90USB162 has only 176 bytes
// of DPRAM (USB buffers) and only endpoints 3 & 4 can double buffer.

#define ENDPOINT0_SIZE     16
#define CDC_ACM_ENDPOINT   2
#define CDC_RX_ENDPOINT    3
#define CDC_TX_ENDPOINT    4
#if defined(__AVR_AT90USB162__)
#define CDC_ACM_SIZE    16
#define CDC_ACM_BUFFER     EP_SINGLE_BUFFER
#define CDC_RX_SIZE     32
#define CDC_RX_BUFFER      EP_DOUBLE_BUFFER
#define CDC_TX_SIZE     32
```

```
#define CDC_TX_BUFFER      EP_DOUBLE_BUFFER
#else
#define CDC_ACM_SIZE    16
#define CDC_ACM_BUFFER     EP_SINGLE_BUFFER
#define CDC_RX_SIZE     64
#define CDC_RX_BUFFER      EP_DOUBLE_BUFFER
#define CDC_TX_SIZE     64
#define CDC_TX_BUFFER      EP_DOUBLE_BUFFER
#endif

static const uint8_t PROGMEM endpoint_config_table[] = {
   0,
   1, EP_TYPE_INTERRUPT_IN,  EP_SIZE(CDC_ACM_SIZE) | CDC_ACM_BUFFER,
   1, EP_TYPE_BULK_OUT,      EP_SIZE(CDC_RX_SIZE) | CDC_RX_BUFFER,
   1, EP_TYPE_BULK_IN,       EP_SIZE(CDC_TX_SIZE) | CDC_TX_BUFFER
};


/**************************************************************************
 *
 *  Descriptor Data
 *
 **************************************************************************/

// Descriptors are the data that your computer reads when it auto-detects
// this USB device (called "enumeration" in USB lingo).  The most commonly
// changed items are editable at the top of this file.  Changing things
// in here should only be done by those who've read chapter 9 of the USB
// spec and relevant portions of any USB class specifications!

const uint8_t PROGMEM device_descriptor[] = {
   18,               // bLength
   1,             // bDescriptorType
   0x00, 0x02,          // bcdUSB
   2,             // bDeviceClass
   0,             // bDeviceSubClass
   0,             // bDeviceProtocol
   ENDPOINT0_SIZE,            // bMaxPacketSize0
   LSB(VENDOR_ID), MSB(VENDOR_ID),     // idVendor
   LSB(PRODUCT_ID), MSB(PRODUCT_ID),   // idProduct
   0x00, 0x01,          // bcdDevice
   1,             // iManufacturer
   2,             // iProduct
   3,             // iSerialNumber
   1              // bNumConfigurations
};

#define CONFIG1_DESC_SIZE (9+9+5+5+4+5+7+9+7+7)
const uint8_t PROGMEM config1_descriptor[CONFIG1_DESC_SIZE] = {
   // configuration descriptor, USB spec 9.6.3, page 264-266, Table 9-10
   9,                // bLength;
   2,             // bDescriptorType;
   LSB(CONFIG1_DESC_SIZE),       // wTotalLength
   MSB(CONFIG1_DESC_SIZE),
   2,             // bNumInterfaces
   1,             // bConfigurationValue
   0,             // iConfiguration
   0xC0,             // bmAttributes
   50,               // bMaxPower
   // interface descriptor, USB spec 9.6.5, page 267-269, Table 9-12
   9,             // bLength
   4,             // bDescriptorType
   0,             // bInterfaceNumber
   0,             // bAlternateSetting
```

```
    1,              // bNumEndpoints
    0x02,              // bInterfaceClass
    0x02,              // bInterfaceSubClass
    0x01,              // bInterfaceProtocol
    0,              // iInterface
    // CDC Header Functional Descriptor, CDC Spec 5.2.3.1, Table 26
    5,              // bFunctionLength
    0x24,              // bDescriptorType
    0x00,              // bDescriptorSubtype
    0x10, 0x01,           // bcdCDC
    // Call Management Functional Descriptor, CDC Spec 5.2.3.2, Table 27
    5,              // bFunctionLength
    0x24,              // bDescriptorType
    0x01,              // bDescriptorSubtype
    0x01,              // bmCapabilities
    1,              // bDataInterface
    // Abstract Control Management Functional Descriptor, CDC Spec 5.2.3.3, Table 28
    4,              // bFunctionLength
    0x24,              // bDescriptorType
    0x02,              // bDescriptorSubtype
    0x06,              // bmCapabilities
    // Union Functional Descriptor, CDC Spec 5.2.3.8, Table 33
    5,              // bFunctionLength
    0x24,              // bDescriptorType
    0x06,              // bDescriptorSubtype
    0,              // bMasterInterface
    1,              // bSlaveInterface0
    // endpoint descriptor, USB spec 9.6.6, page 269-271, Table 9-13
    7,              // bLength
    5,              // bDescriptorType
    CDC_ACM_ENDPOINT | 0x80,      // bEndpointAddress
    0x03,              // bmAttributes (0x03=intr)
    CDC_ACM_SIZE, 0,         // wMaxPacketSize
    64,              // bInterval
    // interface descriptor, USB spec 9.6.5, page 267-269, Table 9-12
    9,              // bLength
    4,              // bDescriptorType
    1,              // bInterfaceNumber
    0,              // bAlternateSetting
    2,              // bNumEndpoints
    0x0A,              // bInterfaceClass
    0x00,              // bInterfaceSubClass
    0x00,              // bInterfaceProtocol
    0,              // iInterface
    // endpoint descriptor, USB spec 9.6.6, page 269-271, Table 9-13
    7,              // bLength
    5,              // bDescriptorType
    CDC_RX_ENDPOINT,         // bEndpointAddress
    0x02,              // bmAttributes (0x02=bulk)
    CDC_RX_SIZE, 0,            // wMaxPacketSize
    0,              // bInterval
    // endpoint descriptor, USB spec 9.6.6, page 269-271, Table 9-13
    7,              // bLength
    5,              // bDescriptorType
    CDC_TX_ENDPOINT | 0x80,       // bEndpointAddress
    0x02,              // bmAttributes (0x02=bulk)
    CDC_TX_SIZE, 0,            // wMaxPacketSize
    0               // bInterval
};

// If you're desperate for a little extra code memory, these strings
// can be completely removed if iManufacturer, iProduct, iSerialNumber
// in the device desciptor are changed to zeros.
struct usb_string_descriptor_struct {
```

```
    uint8_t bLength;
    uint8_t bDescriptorType;
    int16_t wString[];
};
const struct usb_string_descriptor_struct PROGMEM string0 = {
    4,
    3,
    {0x0409}
};
const struct usb_string_descriptor_struct PROGMEM string1 = {
    sizeof(STR_MANUFACTURER),
    3,
    STR_MANUFACTURER
};
const struct usb_string_descriptor_struct PROGMEM string2 = {
    sizeof(STR_PRODUCT),
    3,
    STR_PRODUCT
};
const struct usb_string_descriptor_struct PROGMEM string3 = {
    sizeof(STR_SERIAL_NUMBER),
    3,
    STR_SERIAL_NUMBER
};

// This table defines which descriptor data is sent for each specific
// request from the host (in wValue and wIndex).
const struct descriptor_list_struct {
    uint16_t wValue;
    uint16_t wIndex;
    const uint8_t  *addr;
    uint8_t     length;
} PROGMEM descriptor_list[] = {
    {0x0100, 0x0000, device_descriptor, sizeof(device_descriptor)},
    {0x0200, 0x0000, config1_descriptor, sizeof(config1_descriptor)},
    {0x0300, 0x0000, (const uint8_t *)&string0, 4},
    {0x0301, 0x0409, (const uint8_t *)&string1, sizeof(STR_MANUFACTURER)},
    {0x0302, 0x0409, (const uint8_t *)&string2, sizeof(STR_PRODUCT)},
    {0x0303, 0x0409, (const uint8_t *)&string3, sizeof(STR_SERIAL_NUMBER)}
};
#define NUM_DESC_LIST (sizeof(descriptor_list)/sizeof(struct descriptor_list_struct))


/**************************************************************************
 *
 *  Variables - these are the only non-stack RAM usage
 *
 **************************************************************************/

// zero when we are not configured, non-zero when enumerated
static volatile uint8_t usb_configuration=0;

// the time remaining before we transmit any partially full
// packet, or send a zero length packet.
static volatile uint8_t transmit_flush_timer=0;
static uint8_t transmit_previous_timeout=0;

// serial port settings (baud rate, control signals, etc) set
// by the PC.  These are ignored, but kept in RAM.
static uint8_t cdc_line_coding[7]={0x00, 0xE1, 0x00, 0x00, 0x00, 0x00, 0x08};
static uint8_t cdc_line_rtsdtr=0;


/**************************************************************************
```

```
 *
 *  Public Functions - these are the API intended for the user
 *
 *************************************************************************/

// initialize USB serial
void usb_init(void)
{
   HW_CONFIG();
        USB_FREEZE();             // enable USB
        PLL_CONFIG();             // config PLL, 16 MHz xtal
        while (!(PLLCSR & (1<<PLOCK))) ;  // wait for PLL lock
        USB_CONFIG();             // start USB clock
        UDCON = 0;             // enable attach resistor
   usb_configuration = 0;
   cdc_line_rtsdtr = 0;
        UDIEN = (1<<EORSTE)|(1<<SOFE);
   sei();
}

// return 0 if the USB is not configured, or the configuration
// number selected by the HOST
uint8_t usb_configured(void)
{
   return usb_configuration;
}

// get the next character, or -1 if nothing received
int16_t usb_serial_getchar(void)
{
   uint8_t c, intr_state;

   // interrupts are disabled so these functions can be
   // used from the main program or interrupt context,
   // even both in the same program!
   intr_state = SREG;
   cli();
   if (!usb_configuration) {
      SREG = intr_state;
      return -1;
   }
   UENUM = CDC_RX_ENDPOINT;
   retry:
   c = UEINTX;
   if (!(c & (1<<RWAL))) {
      // no data in buffer
      if (c & (1<<RXOUTI)) {
         UEINTX = 0x6B;
         goto retry;
      }
      SREG = intr_state;
      return -1;
   }
   // take one byte out of the buffer
   c = UEDATX;
   // if buffer completely used, release it
   if (!(UEINTX & (1<<RWAL))) UEINTX = 0x6B;
   SREG = intr_state;
   return c;
}

// number of bytes available in the receive buffer
uint8_t usb_serial_available(void)
{
```

```
    uint8_t n=0, i, intr_state;

    intr_state = SREG;
    cli();
    if (usb_configuration) {
        UENUM = CDC_RX_ENDPOINT;
        n = UEBCLX;
        if (!n) {
            i = UEINTX;
            if (i & (1<<RXOUTI) && !(i & (1<<RWAL))) UEINTX = 0x6B;
        }
    }
    SREG = intr_state;
    return n;
}

// discard any buffered input
void usb_serial_flush_input(void)
{
    uint8_t intr_state;

    if (usb_configuration) {
        intr_state = SREG;
        cli();
        UENUM = CDC_RX_ENDPOINT;
        while ((UEINTX & (1<<RWAL))) {
            UEINTX = 0x6B;
        }
        SREG = intr_state;
    }
}

// transmit a character.  0 returned on success, -1 on error
int8_t usb_serial_putchar(uint8_t c)
{
    uint8_t timeout, intr_state;

    // if we're not online (enumerated and configured), error
    if (!usb_configuration) return -1;
    // interrupts are disabled so these functions can be
    // used from the main program or interrupt context,
    // even both in the same program!
    intr_state = SREG;
    cli();
    UENUM = CDC_TX_ENDPOINT;
    // if we gave up due to timeout before, don't wait again
    if (transmit_previous_timeout) {
        if (!(UEINTX & (1<<RWAL))) {
            SREG = intr_state;
            return -1;
        }
        transmit_previous_timeout = 0;
    }
    // wait for the FIFO to be ready to accept data
    timeout = UDFNUML + TRANSMIT_TIMEOUT;
    while (1) {
        // are we ready to transmit?
        if (UEINTX & (1<<RWAL)) break;
        SREG = intr_state;
        // have we waited too long?  This happens if the user
        // is not running an application that is listening
        if (UDFNUML == timeout) {
            transmit_previous_timeout = 1;
            return -1;
```

```
		}
		// has the USB gone offline?
		if (!usb_configuration) return -1;
		// get ready to try checking again
		intr_state = SREG;
		cli();
		UENUM = CDC_TX_ENDPOINT;
	}
	// actually write the byte into the FIFO
	UEDATX = c;
	// if this completed a packet, transmit it now!
	if (!(UEINTX & (1<<RWAL))) UEINTX = 0x3A;
	transmit_flush_timer = TRANSMIT_FLUSH_TIMEOUT;
	SREG = intr_state;
	return 0;
}


// transmit a character, but do not wait if the buffer is full,
//   0 returned on success, -1 on buffer full or error
int8_t usb_serial_putchar_nowait(uint8_t c)
{
	uint8_t intr_state;

	if (!usb_configuration) return -1;
	intr_state = SREG;
	cli();
	UENUM = CDC_TX_ENDPOINT;
	if (!(UEINTX & (1<<RWAL))) {
		// buffer is full
		SREG = intr_state;
		return -1;
	}
	// actually write the byte into the FIFO
	UEDATX = c;
		// if this completed a packet, transmit it now!
	if (!(UEINTX & (1<<RWAL))) UEINTX = 0x3A;
	transmit_flush_timer = TRANSMIT_FLUSH_TIMEOUT;
	SREG = intr_state;
	return 0;
}

// transmit a buffer.
//  0 returned on success, -1 on error
// This function is optimized for speed!  Each call takes approx 6.1 us overhead
// plus 0.25 us per byte.  12 Mbit/sec USB has 8.67 us per-packet overhead and
// takes 0.67 us per byte.  If called with 64 byte packet-size blocks, this function
// can transmit at full USB speed using 43% CPU time.  The maximum theoretical speed
// is 19 packets per USB frame, or 1216 kbytes/sec.  However, bulk endpoints have the
// lowest priority, so any other USB devices will likely reduce the speed.  Speed
// can also be limited by how quickly the PC-based software reads data, as the host
// controller in the PC will not allocate bandwitdh without a pending read request.
// (thanks to Victor Suarez for testing and feedback and initial code)

int8_t usb_serial_write(const uint8_t *buffer, uint16_t size)
{
	uint8_t timeout, intr_state, write_size;

	// if we're not online (enumerated and configured), error
	if (!usb_configuration) return -1;
	// interrupts are disabled so these functions can be
	// used from the main program or interrupt context,
	// even both in the same program!
	intr_state = SREG;
```

```
	cli();
	UENUM = CDC_TX_ENDPOINT;
	// if we gave up due to timeout before, don't wait again
	if (transmit_previous_timeout) {
		if (!(UEINTX & (1<<RWAL))) {
			SREG = intr_state;
			return -1;
		}
		transmit_previous_timeout = 0;
	}
	// each iteration of this loop transmits a packet
	while (size) {
		// wait for the FIFO to be ready to accept data
		timeout = UDFNUML + TRANSMIT_TIMEOUT;
		while (1) {
			// are we ready to transmit?
			if (UEINTX & (1<<RWAL)) break;
			SREG = intr_state;
			// have we waited too long?  This happens if the user
			// is not running an application that is listening
			if (UDFNUML == timeout) {
				transmit_previous_timeout = 1;
				return -1;
			}
			// has the USB gone offline?
			if (!usb_configuration) return -1;
			// get ready to try checking again
			intr_state = SREG;
			cli();
			UENUM = CDC_TX_ENDPOINT;
		}

		// compute how many bytes will fit into the next packet
		write_size = CDC_TX_SIZE - UEBCLX;
		if (write_size > size) write_size = size;
		size -= write_size;

		// write the packet
		switch (write_size) {
			#if (CDC_TX_SIZE == 64)
			case 64: UEDATX = *buffer++;
			case 63: UEDATX = *buffer++;
			case 62: UEDATX = *buffer++;
			case 61: UEDATX = *buffer++;
			case 60: UEDATX = *buffer++;
			case 59: UEDATX = *buffer++;
			case 58: UEDATX = *buffer++;
			case 57: UEDATX = *buffer++;
			case 56: UEDATX = *buffer++;
			case 55: UEDATX = *buffer++;
			case 54: UEDATX = *buffer++;
			case 53: UEDATX = *buffer++;
			case 52: UEDATX = *buffer++;
			case 51: UEDATX = *buffer++;
			case 50: UEDATX = *buffer++;
			case 49: UEDATX = *buffer++;
			case 48: UEDATX = *buffer++;
			case 47: UEDATX = *buffer++;
			case 46: UEDATX = *buffer++;
			case 45: UEDATX = *buffer++;
			case 44: UEDATX = *buffer++;
			case 43: UEDATX = *buffer++;
			case 42: UEDATX = *buffer++;
			case 41: UEDATX = *buffer++;
```

```
            case 40: UEDATX = *buffer++;
            case 39: UEDATX = *buffer++;
            case 38: UEDATX = *buffer++;
            case 37: UEDATX = *buffer++;
            case 36: UEDATX = *buffer++;
            case 35: UEDATX = *buffer++;
            case 34: UEDATX = *buffer++;
            case 33: UEDATX = *buffer++;
            #endif
            #if (CDC_TX_SIZE >= 32)
            case 32: UEDATX = *buffer++;
            case 31: UEDATX = *buffer++;
            case 30: UEDATX = *buffer++;
            case 29: UEDATX = *buffer++;
            case 28: UEDATX = *buffer++;
            case 27: UEDATX = *buffer++;
            case 26: UEDATX = *buffer++;
            case 25: UEDATX = *buffer++;
            case 24: UEDATX = *buffer++;
            case 23: UEDATX = *buffer++;
            case 22: UEDATX = *buffer++;
            case 21: UEDATX = *buffer++;
            case 20: UEDATX = *buffer++;
            case 19: UEDATX = *buffer++;
            case 18: UEDATX = *buffer++;
            case 17: UEDATX = *buffer++;
            #endif
            #if (CDC_TX_SIZE >= 16)
            case 16: UEDATX = *buffer++;
            case 15: UEDATX = *buffer++;
            case 14: UEDATX = *buffer++;
            case 13: UEDATX = *buffer++;
            case 12: UEDATX = *buffer++;
            case 11: UEDATX = *buffer++;
            case 10: UEDATX = *buffer++;
            case  9: UEDATX = *buffer++;
            #endif
            case  8: UEDATX = *buffer++;
            case  7: UEDATX = *buffer++;
            case  6: UEDATX = *buffer++;
            case  5: UEDATX = *buffer++;
            case  4: UEDATX = *buffer++;
            case  3: UEDATX = *buffer++;
            case  2: UEDATX = *buffer++;
            default:
            case  1: UEDATX = *buffer++;
            case  0: break;
        }
        // if this completed a packet, transmit it now!
        if (!(UEINTX & (1<<RWAL))) UEINTX = 0x3A;
        transmit_flush_timer = TRANSMIT_FLUSH_TIMEOUT;
        SREG = intr_state;
    }
    return 0;
}


// immediately transmit any buffered output.
// This doesn't actually transmit the data - that is impossible!
// USB devices only transmit when the host allows, so the best
// we can do is release the FIFO buffer for when the host wants it
void usb_serial_flush_output(void)
{
```

```
	uint8_t intr_state;

	intr_state = SREG;
	cli();
	if (transmit_flush_timer) {
		UENUM = CDC_TX_ENDPOINT;
		UEINTX = 0x3A;
		transmit_flush_timer = 0;
	}
	SREG = intr_state;
}

// functions to read the various async serial settings.  These
// aren't actually used by USB at all (communication is always
// at full USB speed), but they are set by the host so we can
// set them properly if we're converting the USB to a real serial
// communication
uint32_t usb_serial_get_baud(void)
{
	return *(uint32_t *)cdc_line_coding;
}
uint8_t usb_serial_get_stopbits(void)
{
	return cdc_line_coding[4];
}
uint8_t usb_serial_get_paritytype(void)
{
	return cdc_line_coding[5];
}
uint8_t usb_serial_get_numbits(void)
{
	return cdc_line_coding[6];
}
uint8_t usb_serial_get_control(void)
{
	return cdc_line_rtsdtr;
}
// write the control signals, DCD, DSR, RI, etc
// There is no CTS signal.  If software on the host has transmitted
// data to you but you haven't been calling the getchar function,
// it remains buffered (either here or on the host) and can not be
// lost because you weren't listening at the right time, like it
// would in real serial communication.
int8_t usb_serial_set_control(uint8_t signals)
{
	uint8_t intr_state;

	intr_state = SREG;
	cli();
	if (!usb_configuration) {
		// we're not enumerated/configured
		SREG = intr_state;
		return -1;
	}

	UENUM = CDC_ACM_ENDPOINT;
	if (!(UEINTX & (1<<RWAL))) {
		// unable to write
		// TODO; should this try to abort the previously
		// buffered message??
		SREG = intr_state;
		return -1;
	}
	UEDATX = 0xA1;
```

```
   UEDATX = 0x20;
   UEDATX = 0;
   UEDATX = 0;
   UEDATX = 0; // 0 seems to work nicely.  what if this is 1??
   UEDATX = 0;
   UEDATX = 1;
   UEDATX = 0;
   UEDATX = signals;
   UEINTX = 0x3A;
   SREG = intr_state;
   return 0;
}



/**************************************************************************
 *
 *  Private Functions - not intended for general user consumption....
 *
 **************************************************************************/


// USB Device Interrupt - handle all device-level events
// the transmit buffer flushing is triggered by the start of frame
//
ISR(USB_GEN_vect)
{
   uint8_t intbits, t;

        intbits = UDINT;
        UDINT = 0;
        if (intbits & (1<<EORSTI)) {
      UENUM = 0;
      UECONX = 1;
      UECFG0X = EP_TYPE_CONTROL;
      UECFG1X = EP_SIZE(ENDPOINT0_SIZE) | EP_SINGLE_BUFFER;
      UEIENX = (1<<RXSTPE);
      usb_configuration = 0;
      cdc_line_rtsdtr = 0;
        }
   if (intbits & (1<<SOFI)) {
      if (usb_configuration) {
         t = transmit_flush_timer;
         if (t) {
            transmit_flush_timer = --t;
            if (!t) {
               UENUM = CDC_TX_ENDPOINT;
               UEINTX = 0x3A;
            }
         }
      }
   }
}


// Misc functions to wait for ready and send/receive packets
static inline void usb_wait_in_ready(void)
{
   while (!(UEINTX & (1<<TXINI))) ;
}
static inline void usb_send_in(void)
{
   UEINTX = ~(1<<TXINI);
}
```

```
static inline void usb_wait_receive_out(void)
{
   while (!(UEINTX & (1<<RXOUTI))) ;
}
static inline void usb_ack_out(void)
{
   UEINTX = ~(1<<RXOUTI);
}



// USB Endpoint Interrupt - endpoint 0 is handled here.  The
// other endpoints are manipulated by the user-callable
// functions, and the start-of-frame interrupt.
//
ISR(USB_COM_vect)
{
        uint8_t intbits;
   const uint8_t *list;
        const uint8_t *cfg;
   uint8_t i, n, len, en;
   uint8_t *p;
   uint8_t bmRequestType;
   uint8_t bRequest;
   uint16_t wValue;
   uint16_t wIndex;
   uint16_t wLength;
   uint16_t desc_val;
   const uint8_t *desc_addr;
   uint8_t  desc_length;

        UENUM = 0;
        intbits = UEINTX;
        if (intbits & (1<<RXSTPI)) {
                bmRequestType = UEDATX;
                bRequest = UEDATX;
                wValue = UEDATX;
                wValue |= (UEDATX << 8);
                wIndex = UEDATX;
                wIndex |= (UEDATX << 8);
                wLength = UEDATX;
                wLength |= (UEDATX << 8);
                UEINTX = ~((1<<RXSTPI) | (1<<RXOUTI) | (1<<TXINI));
                if (bRequest == GET_DESCRIPTOR) {
         list = (const uint8_t *)descriptor_list;
         for (i=0; ; i++) {
            if (i >= NUM_DESC_LIST) {
               UECONX = (1<<STALLRQ)|(1<<EPEN);  //stall
               return;
            }
            desc_val = pgm_read_word(list);
            if (desc_val != wValue) {
               list += sizeof(struct descriptor_list_struct);
               continue;
            }
            list += 2;
            desc_val = pgm_read_word(list);
            if (desc_val != wIndex) {
               list += sizeof(struct descriptor_list_struct)-2;
               continue;
            }
            list += 2;
            desc_addr = (const uint8_t *)pgm_read_word(list);
            list += 2;
```

```
            desc_length = pgm_read_byte(list);
            break;
         }
         len = (wLength < 256) ? wLength : 255;
         if (len > desc_length) len = desc_length;
         do {
            // wait for host ready for IN packet
            do {
               i = UEINTX;
            } while (!(i & ((1<<TXINI)|(1<<RXOUTI))));
            if (i & (1<<RXOUTI)) return;  // abort
            // send IN packet
            n = len < ENDPOINT0_SIZE ? len : ENDPOINT0_SIZE;
            for (i = n; i; i--) {
               UEDATX = pgm_read_byte(desc_addr++);
            }
            len -= n;
            usb_send_in();
         } while (len || n == ENDPOINT0_SIZE);
         return;
                }
      if (bRequest == SET_ADDRESS) {
         usb_send_in();
         usb_wait_in_ready();
         UDADDR = wValue | (1<<ADDEN);
         return;
      }
      if (bRequest == SET_CONFIGURATION && bmRequestType == 0) {
         usb_configuration = wValue;
         cdc_line_rtsdtr = 0;
         transmit_flush_timer = 0;
         usb_send_in();
         cfg = endpoint_config_table;
         for (i=1; i<5; i++) {
            UENUM = i;
            en = pgm_read_byte(cfg++);
            UECONX = en;
            if (en) {
               UECFG0X = pgm_read_byte(cfg++);
               UECFG1X = pgm_read_byte(cfg++);
            }
         }
            UERST = 0x1E;
            UERST = 0;
         return;
      }
      if (bRequest == GET_CONFIGURATION && bmRequestType == 0x80) {
         usb_wait_in_ready();
         UEDATX = usb_configuration;
         usb_send_in();
         return;
      }
      if (bRequest == CDC_GET_LINE_CODING && bmRequestType == 0xA1) {
         usb_wait_in_ready();
         p = cdc_line_coding;
         for (i=0; i<7; i++) {
            UEDATX = *p++;
         }
         usb_send_in();
         return;
      }
      if (bRequest == CDC_SET_LINE_CODING && bmRequestType == 0x21) {
         usb_wait_receive_out();
         p = cdc_line_coding;
```

```
         for (i=0; i<7; i++) {
            *p++ = UEDATX;
         }
         usb_ack_out();
         usb_send_in();
         return;
      }
      if (bRequest == CDC_SET_CONTROL_LINE_STATE && bmRequestType == 0x21) {
         cdc_line_rtsdtr = wValue;
         usb_wait_in_ready();
         usb_send_in();
         return;
      }
      if (bRequest == GET_STATUS) {
         usb_wait_in_ready();
         i = 0;
         #ifdef SUPPORT_ENDPOINT_HALT
         if (bmRequestType == 0x82) {
            UENUM = wIndex;
            if (UECONX & (1<<STALLRQ)) i = 1;
            UENUM = 0;
         }
         #endif
         UEDATX = i;
         UEDATX = 0;
         usb_send_in();
         return;
      }
      #ifdef SUPPORT_ENDPOINT_HALT
      if ((bRequest == CLEAR_FEATURE || bRequest == SET_FEATURE)
        && bmRequestType == 0x02 && wValue == 0) {
         i = wIndex & 0x7F;
         if (i >= 1 && i <= MAX_ENDPOINT) {
            usb_send_in();
            UENUM = i;
            if (bRequest == SET_FEATURE) {
               UECONX = (1<<STALLRQ)|(1<<EPEN);
            } else {
               UECONX = (1<<STALLRQC)|(1<<RSTDT)|(1<<EPEN);
               UERST = (1 << i);
               UERST = 0;
            }
            return;
         }
      }
      #endif
        }
   UECONX = (1<<STALLRQ) | (1<<EPEN);  // stall
}
```

APPENDIX B
ARRAY CONTROL SERVER SOURCE CODE

The following is a reproduction of the Python source code of the array control server software. The source files are grouped into sections by the logical nature of their function. These sections are as follows:

- **Main Code**: The main Python class containing the main loop of the program, database connection & initialization functions, thermal messaging logic, and UI command parsing logic
- **User Interface Server**: The Python class responsible for managing network communication with user interface clients
- **Antenna Module Servers**: The Python classes for managing the modular antenna controller network, including the module connection announcement server class and the module command handler class

B.1 Main Code

array_server.py

```
#! /usr/bin/env python

import sys
import datetime
import socket
import time
import threading
from multiprocessing import Queue
import select
import apsw
import argparse
import string

import UIServer
import ModAnnounceServer
import ModuleHandler

__version__ = "0.1"


UI_PORT = 9500    # Port to listen for UI client connections

MOD_ANNOUNCE_PORT = 9100   # Port to listen for module startup announcements
MOD_CONNECT_PORT = 9101    # Port to connect to modules to send commands

delim = ";"

printable = string.printable[0:95]
asciiDict = dict()
for char in printable:
   asciiDict[char] = ord(char)


argParser = argparse.ArgumentParser(
      description='Aperskin Array UI/Module Control Server')
argParser.add_argument(
      '-db','--database',help='Path to module database',default='./modules.db')
argParser.add_argument(
      '-initdb',help='Initialize module database',action='store_true')
argParser.add_argument(
      '-log','--logfile',help='Path to log file',default='./server.log')

# Create tuple of set & lock to store connected module addresses
moduleSet = (set(),threading.Lock())
#Create tuple of dictionary & lock to map module addresses to numeric IDs
moduleDict = (dict(),threading.Lock())
uiTXQueue = Queue()
uiRXQueue = Queue()

#class commandParser(threading.Thread):
#  def __init__(self):
#
#  def parseCommand(command):

class Logger(object):
   def __init__(self,filename):
      self.terminal = sys.stdout
      self.log = open(filename,"a")

   def write(self,msg):
```

```
        self.terminal.write(msg)
        if msg != "\n":
            self.log.write(str(datetime.datetime.now()).split(".")[0] + " " + msg)
        else:
            self.log.write(msg)
        self.log.flush()


# Create database schema error class to raise
# when database's modules table is improperly formatted
class DatabaseSchemaError(Exception):
    def __init__(self,value):
        self.value = value
    def __str__(self):
        return repr(self.value)

if __name__ == '__main__':

    arguments = argParser.parse_args()  # Get command line arguments

    sys.stdout = Logger(arguments.logfile)

    dbConn = apsw.Connection(arguments.database) # Connect to module database
    dbCursor = dbConn.cursor()

    # Initialize modules table if argument passed
    if arguments.initdb:
        schema = '''
            CREATE TABLE modules (
                serial      TEXT PRIMARY KEY,
                id    INTEGER UNIQUE,
                ip    TEXT,
                connected   INTEGER,
                ts    TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
            )
            '''
        dbCursor.execute(schema)

    # Check if modules table is present, create it if not
    dbCursor.execute("SELECT name FROM sqlite_master WHERE type='table' AND name='modules'")
    if dbCursor.fetchall() == []:
        print "WARNING: Database not initialized"
        print "Initializing empty table for modules"

        schema = '''
            CREATE TABLE modules (
                serial      TEXT PRIMARY KEY,
                id    INTEGER UNIQUE,
                ip    TEXT,
                connected   INTEGER,
                ts    TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
            )
            '''
        dbCursor.execute(schema)

    # Check if modules table schema is correct
    schema = list(dbCursor.execute("PRAGMA table_info('modules')"))
    if schema != [ (0, u'serial',    u'TEXT', 0, None,       1),
          (1, u'id',     u'INTEGER', 0, None,       0),
          (2, u'ip',     u'TEXT', 0, None,       0),
          (3, u'connected', u'INTEGER', 0, None,       0),
          (4, u'ts',     u'TIMESTAMP', 1, u'CURRENT_TIMESTAMP',   0)]:

        raise DatabaseSchemaError("modules table incorrectly formatted")
```

```python

    # Instantiate module announcement server
    # Daemonizing the child thread lets us terminate it by ending the parent without joining
    myModAnnounceServer = ModAnnounceServer.ModAnnounceServer(MOD_ANNOUNCE_PORT, dbConn)
    myModAnnounceServer.setDaemon(True)
    myModAnnounceServer.start()

    # Instantiate UI connection server
    myUIServer = UIServer.UIServer(dbConn, UI_PORT, uiTXQueue, uiRXQueue)
    myUIServer.setDaemon(True)
    myUIServer.start()

class ThermalMessager:
   def __init__(self, dbConn, port, UIClient, charDict,
            idList, message, charTime, highTemp, lowTemp):
      self.dbConn = dbConn
      self.port = port
      self.UIClient = UIClient
      self.charDict = charDict
      self.idList = idList
      self.message = message
      self.charTime = charTime
      self.highTemp = highTemp
      self.lowTemp = lowTemp

      self.tempCommand = "SETTEMP {0}\r"
      self.curChar = 0


   def send(self):
      for char in message:
         if char not in printable:
            self.status = 1
            return self.status
         else:
            self.next_call = time.time()
            self.run = True
            self.__nextChar__()
            self.status = 0
            return self.status

   def __nextChar__(self):
      if self.run:
         char = self.message[self.curChar]
         ascValue = self.charDict[char]
         toHighTemp = []
         toLowTemp = []
         for i in range(len(idList)):
            if bool(ascValue & (1 << i)):
               toHighTemp.append(self.idList[i])
            else:
               toLowTemp.append(self.idList[i])

         toHighResponses = ModuleHandler.sendCommand(
            self.dbConn, self.port, toHighTemp,
            self.tempCommand.format(str(self.highTemp)))
         toLowResponses = ModuleHandler.sendCommand(
            self.dbConn, self.port, toLowTemp,
            self.tempCommand.format(str(self.lowTemp)))

         modResponses = toHighResponses
         for response in toLowResponses:
            modResponses.append(response)
```

```
            print str(modResponses)
            for response in modResponses:
                if response[1] != 0 or response[2] == "ERROR":
                    uiTXQueue.put((self.UIClient,
                        "ERROR: Sending character \'" +str(char)+
                        "\' to array:\r\n"))
                    uiTXQueue.put((self.UIClient,
                        "Module "+str(response[0]) +
                        "Status Code:" + str(response[1]) +
                        ":" + str(response[2]).strip() + "\r\n"))
            uiTXQueue.put((self.UIClient,
                "Sent character \'" + str(char) + "\' to array\r\n"))

            self.curChar += 1
            if (self.curChar >= len(self.message)):
                self.run = False
            else:
                self.next_call += self.charTime
                uiTXQueue.put((self.UIClient, "Waiting "+str(self.charTime)+" seconds\r\n"))
                threading.Timer(self.next_call - time.time(), self.__nextChar__).start()

while True:
    (UIClient,UIcommand) = uiRXQueue.get()
    try:
        commandScope,command = UIcommand.split(delim,1)
    except ValueError:
        uiTXQueue.put((UIClient,'ERROR: Invalid syntax\r\n'))
    else:
        if commandScope.lower() == "array":
            print "Received array-wide command: " + str(command)
            moduleIDs = list(
                dbCursor.execute("SELECT id FROM modules WHERE connected=1 ORDER BY id"))

            moduleResponses = ModuleHandler.sendCommand(
                dbConn,MOD_CONNECT_PORT,moduleIDs,command)

            for response in moduleResponses:
                uiTXQueue.put((UIClient,str(response[0]) +
                    ":" + str(response[1]) +
                    ":" + str(response[2]).strip() + "\r\n"))

        elif commandScope.lower() == "module":
            try:
                idList,command = command.split(delim,1)

                # Convert comma-separated string to list of IDs
                idList = list(idList.split(","))
            except ValueError as e:
                uiTXQueue.put((UIClient, "ERROR: Invalid Syntax\r\n"))
            else:
                print "Received command for modules " + str(idList) + " : " + command


                moduleResponses = ModuleHandler.sendCommand(
                    dbConn,MOD_CONNECT_PORT,idList,command)

                for response in moduleResponses:
                    uiTXQueue.put((UIClient,str(response[0]) +
                        ":" + str(response[1]) +
                        ":" + str(response[2]).strip() + "\r\n"))


        elif commandScope.lower() == "irmsg":
            print "Received thermal message command: " + command
```

```
            try:
               idList,command = command.split(delim,1)
               message,delay = command.split(" ",1)
               idList = list(idList.split(","))
               delay = float(delay)
            except ValueError as e:
               uiTXQueue.put((UIClient, "ERROR: Invalid Syntax\r\n"))
            else:
               myThermalMessager = ThermalMessager(dbConn,MOD_CONNECT_PORT,UIClient,
                  asciiDict,idList,message,delay,35.0,20.0)
               status = myThermalMessager.send()
               if status != 0:
                  uiTXQueue.put((UIClient, "ERROR: Invalid Message\r\n"))

         elif commandScope.lower() == "server":
            print "Received server command: " + command

            try:
               command,argument = command.split(delim,1)
            except ValueError as e:
               pass

            # auto-ID: associate numeric IDs to connected modules based on connection order
            if command.lower() == "autoid":
               serialList = list(dbCursor.execute(
                  "SELECT serial FROM modules WHERE connected=1 ORDER BY ts ASC"))
               for i in range(len(serialList)):
                  query = "UPDATE OR IGNORE modules SET id=:id WHERE serial=:serial"
                  dbCursor.execute(query,{"id": i, "serial": serialList[i][0]})
               uiTXQueue.put((UIClient,"Module IDs assigned\r\n"))

            elif command.lower() == "setid":
               try:
                  serial,id = argument.split(" ",1)
               except ValueError as e:
                  uiTXQueue.put((UIClient, "ERROR: Invalid Syntax\r\n"))
               else:
                  if id == "None":
                     id = None
                  try:
                     id = int(id)

                     serialList = list(dbCursor.execute(
                        "SELECT serial FROM modules WHERE id=:id",{"id":id}))
                     if len(serialList) > 0:
                        uiTXQueue.put((UIClient,
                           "ID "+str(id)+" already assigned to serial "+
                           str(serialList)+"\r\n"))
                     else:
                        query = '''UPDATE OR IGNORE modules SET
                           id=:id WHERE serial LIKE :serial; '''
                        query += "SELECT serial FROM modules WHERE serial LIKE :serial"
                        serials = list(
                           dbCursor.execute(query,{"id": id, "serial": "%"+str(serial)}))
                        uiTXQueue.put((UIClient,
                           "ID "+str(id)+
                           " assigned to serial "+str(serials)+"\r\n"))
                  except ValueError as e:
                     uiTXQueue.put((UIClient, "ERROR: Invalid Syntax\r\n"))
                  except TypeError as e:
                     pass

            elif command.lower() == "clearids":
               dbCursor.execute("UPDATE OR IGNORE modules SET id=:id",{"id": None})
```

```
               uiTXQueue.put((UIClient,"Module IDs cleared\r\n"))

            elif command.lower() == "unsetid":
               try:
                  id = int(argument)
               except ValueError as e:
                  uiTXQueue.put((UIClient, "ERROR: Invalid Syntax\r\n"))
               else:
                  query = "UPDATE OR IGNORE modules SET id=:idNew WHERE id=:idOld"
                  dbCursor.execute(query,{"idNew": None, "idOld": id})
                  uiTXQueue.put((UIClient, "ID "+str(id)+" unset\r\n"))


            elif command.lower() == "status":
                        greeting = "Connected to Array Server v." +
                           str(__version__) + "\r\n"
                        connectedModules = list(dbCursor.execute(
                          '''SELECT serial,id,ip
                              FROM modules WHERE connected=1 ORDER BY id'''))
                         greeting += "Modules Connected: " +
                           str(len(connectedModules)) +"\r\n"
                         greeting += "SERIAL:\t\tID:\tIP ADDRESS:\r\n"
                         for (serial,id,ip) in connectedModules:
                                 greeting += str(serial) +
                                    "\t" + str(id) +
                                    "\t" + str(ip) + "\r\n"
                         uiTXQueue.put((UIClient,greeting))

            elif command.lower() == "listall":
               allModules = list(dbCursor.execute(
                  "SELECT serial,id,ip,connected,ts FROM modules"))
               response = "Known modules: " + str(len(allModules)) + "\r\n"
               response += "SERIAL:\t\tID:\tIP ADDRESS:\tCONNECTED:\tLAST UPDATED:\r\n"
               for (serial,id,ip,connected,ts) in allModules:
                  response += str(serial) + "\t" + str(id) + "\t" + str(ip) + "\t"
                  if connected == 1:
                     response += "Yes"
                  else:
                     response += "No"
                  response += "\t\t" + str(ts) + "\r\n"
               uiTXQueue.put((UIClient,response))

            elif command.lower() == "test":
               uiTXQueue.put((UIClient,"25\r\n"))

            else:
               uiTXQueue.put((UIClient,"ERROR: Invalid Syntax\r\n"))
```

B.2 User Interface Server

UIServer.py

```
#! /usr/bin/env python

import socket
import time
import threading
from multiprocessing import Queue
import select

__version__ = "0.1"

class UIServer(threading.Thread):
   def __init__(self,dbConn,ui_port,TXQueue,RXQueue):
```

```
      threading.Thread.__init__(self)
      self.serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
      self.serversocket.bind(('', ui_port))
      self.serversocket.listen(5)
      print "Listening for UI client connections on " + str(ui_port)

      self.dbConn = dbConn

      self.TXQueue = TXQueue
      self.RXQueue = RXQueue

   def run(self):
      while True:
         (clientsocket,clientaddress) = self.serversocket.accept()
         newUIHandler = UIHandler(self.dbConn,(clientsocket,clientaddress),
               self.TXQueue,self.RXQueue)
         newUIHandler.start()


class UIHandler(threading.Thread):
   def __init__(self,dbConn,(clientsocket,address),TXQueue,RXQueue):
      threading.Thread.__init__(self)
      self.clientsocket = clientsocket
      self.clientaddress = address
      self.buffersize = 4096

      self.TXQueue = TXQueue
      self.RXQueue = RXQueue
      self.dbConn = dbConn

      print "UI client connected at " + str(self.clientaddress)

   def run(self):
      running = True
      self.greet(self.clientsocket)
      while running:
         (readable,writeable,exceptable) =
            select.select([self.clientsocket,self.TXQueue._reader],[],[],10)
         for source in readable:
            if source is self.clientsocket:
               inData = self.clientsocket.recv(self.buffersize)
               if inData:
                  if inData.strip().lower() == "quit":
                     self.clientsocket.close()
                     print "UI client quit"
                     running = False
                     break
                  else:
                     print "UI client sent: " + repr(inData)
                     self.RXQueue.put((self.clientaddress,inData.strip()))
               elif not inData:
                  print "UI client disconnected at: " + str(self.clientaddress)
                  running = False
                  break
            elif source is self.TXQueue._reader:
               (client,outData) = self.TXQueue.get()
               if client == self.clientaddress:
                  #print "Sending to UI client: " + str(outData)
                  self.clientsocket.sendall(outData)
               elif client != self.clientaddress:
                  self.TXQueue.put((client,outData))
                  time.sleep(0.001)

   def greet(self,clientsocket):
```

```
      greeting = "Connected to Array Server v." + str(__version__) + "\r\n"

      dbCursor = self.dbConn.cursor()
      connectedModules = list(
         dbCursor.execute("SELECT serial,id,ip FROM modules WHERE connected=1 ORDER BY id"))

      greeting += "Modules Connected: " + str(len(connectedModules)) +"\r\n"
      greeting += "SERIAL:\t\tID:\tIP ADDRESS:\r\n"
      for (serial,id,ip) in connectedModules:
         greeting += str(serial) + "\t" + str(id) + "\t" + str(ip) + "\r\n"
      clientsocket.sendall(greeting)
```

B.3 Antenna Module Servers

ModAnnounceServer.py

```
#! /usr/bin/env python

import socket
import threading
import apsw

class ModAnnounceServer(threading.Thread):

   def __init__(self, port, dbConnection):
      threading.Thread.__init__(self)
      self.serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
      self.serversocket.bind(('', port))
      self.serversocket.listen(5)
      print "Listening for module announcement connections on " + str(port)

      self.dbConnection = dbConnection

   def run(self):
      running = True
      while running:
         (clientsocket, clientaddress) = self.serversocket.accept()
         newAnnounceHandler =
            AnnounceHandler((clientsocket,clientaddress),self.dbConnection)
         newAnnounceHandler.start()


class AnnounceHandler(threading.Thread):

   def __init__(self, (clientsocket,address), dbConnection):
      threading.Thread.__init__(self)
      self.clientsocket = clientsocket
      self.clientaddress = address
      self.buffersize = 4096

      self.dbConnection = dbConnection

      print 'Module connected at ' + str(self.clientaddress)

   def run(self):
      running = True
      while running:
         inData = self.clientsocket.recv(self.buffersize)
         try:
            (serial,status) = inData.strip().split(' ',1)
         except ValueError:              #If split failed, syntax was wrong
            print 'Module response invalid from ' + str(self.clientaddress)
         else:
```

```
                if status == "READY":
                    (address,port) = self.clientaddress

                    # Add or update module's entry in module database
                    dbCursor = self.dbConnection.cursor()

                    query = '''INSERT OR IGNORE INTO modules (serial, ip, connected)
                        VALUES (:serial, :ip, :connected);
                        UPDATE modules SET
                        serial=:serial,
                        ip=:ip,
                        connected=:connected,
                        ts=CURRENT_TIMESTAMP
                        WHERE serial=:serial;
                        '''

                    dbCursor.execute(query,{"serial": serial, "ip": address, "connected": 1})

                    print 'Module '+ str(serial) + ' ready at ' + str(self.clientaddress)

                    #self.clientsocket.shutdown(socket.SHUT_RDWR)
                    self.clientsocket.close()
                    running = False
                else:
                    print 'Module not ready at ' + str(self.clientaddress)
                    #self.clientsocket.shutdown(socket.SHUT_RDWR)
                    self.clientsocket.close()
                    running = False
```

ModuleHandler.py

```
# Module handler to dispatch command string to a connected module and retrieve its response
# Prunes modules from connected module list on connect failure

import socket
import threading
import apsw

#

class ModuleHandler(threading.Thread):
    def __init__(self,dbConnection,port,moduleID,command):
        threading.Thread.__init__(self)
        self.dbConnection = dbConnection
        self.port = port
        self.moduleID = moduleID
        self.command = command
        self.buffersize = 4096

        self.modulesocket = socket.socket(socket.AF_INET,socket.SOCK_STREAM)
        self.modulesocket.settimeout(10)
        self.retries = 3

    def run(self):
        dbCursor = self.dbConnection.cursor()
        allModuleIDs = list(dbCursor.execute("SELECT id FROM modules"))

        if (int(self.moduleID),) in allModuleIDs:

            (self.moduleSerial,self.moduleIP) = list(dbCursor.execute(
                "SELECT serial,ip FROM modules WHERE id=:id",{"id":self.moduleID}))[0]

            for attempt in range(1,self.retries+1):
                try:
```

```python
                    print "Connecting to module: " + str(self.moduleID) +
                        " at " + str(self.moduleIP) +
                        ", attempt " + str(attempt)
                    # Connect to module
                    self.modulesocket.connect((self.moduleIP,self.port))
                    # Send command to module with terminating CR
                    self.modulesocket.send(self.command+'\r')

                except socket.error:                # Catch connection or send error
                    print "Error sending command to module: " +
                        str(self.moduleID) + " at " + str(self.moduleIP)
                    self.status = 1
                    self.response = ''
                    # If connect fails self.retries times, module is dead
                    if attempt == self.retries:
                        dbCursor.execute(
                            "UPDATE modules SET connected=0,ts=CURRENT_TIMESTAMP WHERE id=:id",
                            {"id":self.moduleID})
                else:
                    try:
                        inData = ''
                        inData = self.modulesocket.recv(self.buffersize)   # Get module's response
                    except socket.timeout:
                        print "Module timed out: "  +
                            str(self.moduleID) + " at " +
                            str(self.moduleIP)
                    finally:
                        self.modulesocket.shutdown(socket.SHUT_RDWR)

                    if inData:
                        print "Got response from module: "  +
                            str(self.moduleID) + " at " + str(self.moduleIP)
                        self.status = 0
                        self.response = inData

                        dbCursor.execute(
                            "UPDATE modules SET connected=1,ts=CURRENT_TIMESTAMP WHERE id=:id",
                            {"id:":self.moduleID})
                    else:
                        print "No response from module: "  +
                            str(self.moduleID) + " at " + str(self.moduleIP)
                        self.status = 2
                        self.response = ''
                if self.status == 0:
                    break

        else:
            print "Module ID does not exist: " + repr(self.moduleID)
            self.status = 3      # Module ID isn't assigned... don't attempt to send
            self.response = ''

        self.modulesocket.close()

def sendCommand(dbConnection,port,moduleIDs,command):
    moduleThreads = []
    moduleResponses = []

    for (moduleID,) in moduleIDs:
        thread = ModuleHandler(dbConnection,port,moduleID,command)
        thread.start()
        moduleThreads.append(thread)


    for thread in moduleThreads:
```

```
        thread.join()
        moduleResponses.append((thread.moduleID,thread.status,thread.response))

    return moduleResponses
```

APPENDIX C
MODULE COMMAND REFERENCE

Table C.1 lists the text commands supported by the MCB firmware. The commands are grouped in the first two columns by command class and the MCB system with which the commands interact. The last four columns contain the command and the required, space-separated arguments for that command. For text arguments, the available argument selections for a given command are denoted by a comma separated list enclosed in square brackets. For numerical arguments, the type and unit of the argument are denoted in angle brackets, and the allowable range further denoted within square brackets. Multi-part arguments (i.e. those for the `SETBEAM` and `SETDAC` commands) are separated by spaces unless otherwise noted in the table (i.e. argument 2 of the `SETVALVEPARAM` command requires exactly 3 comma separated pulse width values). Commands require *all* of their listed arguments with the following exceptions:

- `SETPID`: When argument 1 is `WRITE`, argument 2 is unnecessary
- `SETPUMPPARAM`: When argument 1 is `WRITE`, arguments 2 & 3 are unnecessary
- `SETVALVEPARAM`: When argument 1 is `WRITE`, argument 2 is unnecessary

Finally, the `SETPUMP` and `SETVALVE` commands are unique in that arguments 1 & 2 can either be a single item, or a comma separated list of pump/valve numbers and a comma separated list of directions/positions. This allows a single command to be issued to control any subset of the pumps or valves, up to and including the entire set, on a given module.

Table C.1: Modular controller board command reference

Class	System	Command	Argument 1	Argument 2	Argument 3
GET	Thermal Control	GETSENSE			
		GETTEMP	[TEC,SINK,MOD,TGT]		
		GETMODE			
		GETPID			
	Antenna Configuration	GETPUMP			
		GETVALVE			
SET	Thermal Control	SETSENSE	[TEC,MODULE]		
		SETTEMP	<Target Temp (C) [0.0-50.0]>		
		SETMODE	[AUTO,MAN,OFF]		
		SETPID	[KP,KI,KD]/[WRITE]	<P/I/D Gain [0-65535]>	
		SETTEC	[HEAT,COOL]	<Pulse Width [0-1023]>	
	Antenna Configuration	SETPUMP	<Pump No(s) [1-4]>	<Dir(s) [IN,OUT,OFF]>	
		SETVALVE	<Valve No(s) [1-8]>	<Posit(s) [0,1,2]>	
		SETSERVOPW	<Valve No [1-8]>	<PW [0-255]>	
		SETPUMPPARAM	<Pump No [1-4]>/[WRITE]	[SPEED,DELAY]	<PW [0-1023]/Delay (ms) [0-65535]>
		SETVALVEPARAM	<Valve No [1-8]>/[WRITE]	<PW0>,<PW1>,<PW2>	
	Array Beam Steering	SETBEAM	<Theta (°)><Phi (°)>		
		SETDAC	<DAC No [1-8]><DAC Word [0-65535]>		
Emergency		SCRAM	[OFF]		
Module Polling		READY?			

APPENDIX D
MODULAR CONTROL BOARD DESIGN & PCB LAYOUT

The following is a reproduction of the schematic diagrams and PCB layout of the modular control board as designed in *KiCAD*. Figs. D.1 – D.4 show the schematic diagram, grouped into major subsystems. Fig. D.5 shows the bottom copper layer of the PCB layout. Fig. D.6 shows the top copper layer. Fig. D.7 shows the silkscreen layer with printed component designators and outlines. Finally, Fig. D.8 shows the soldermask layer, with colored areas representing the soldermask keepout areas around component pads.

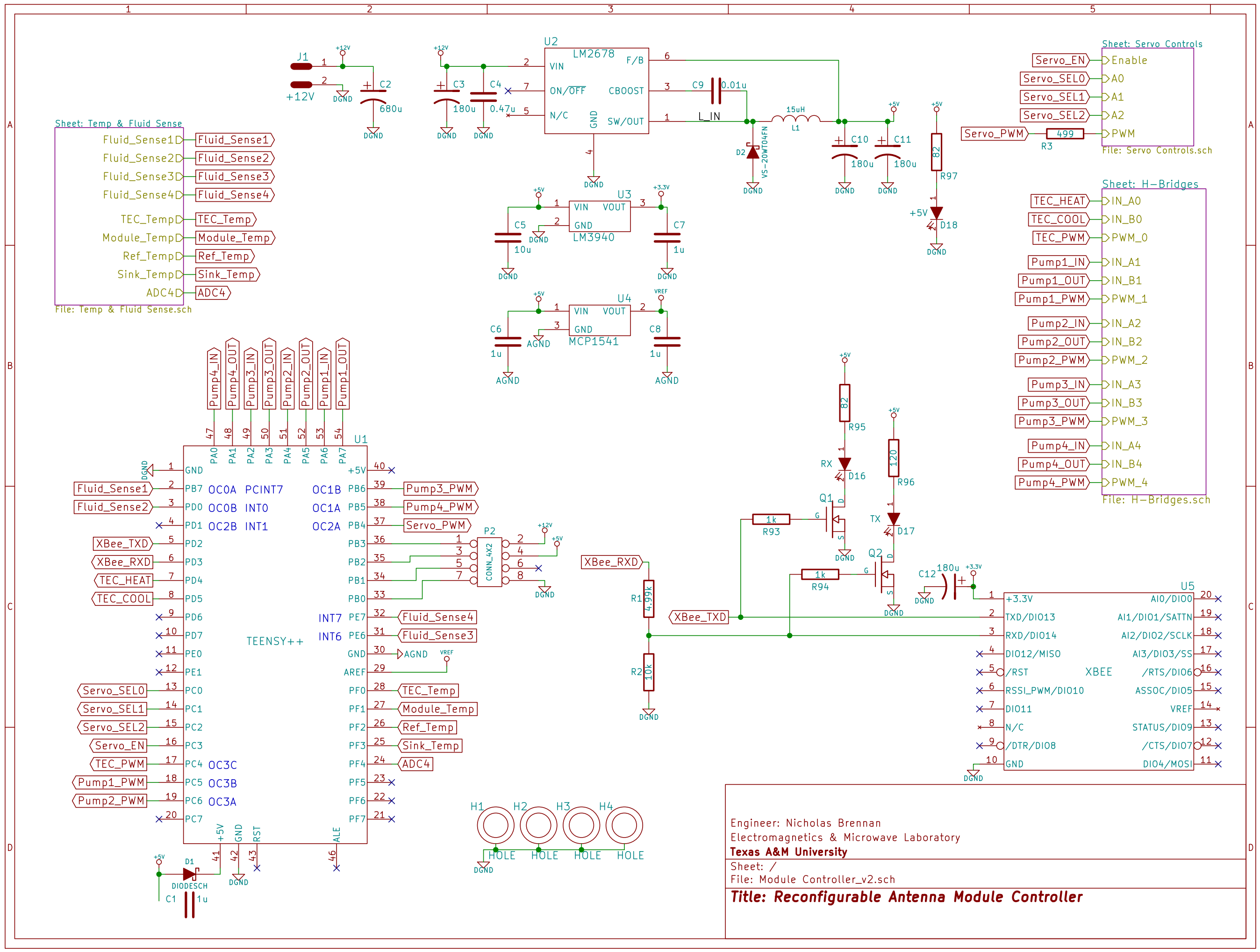


Fig. D.1: Modular control board: main schematic

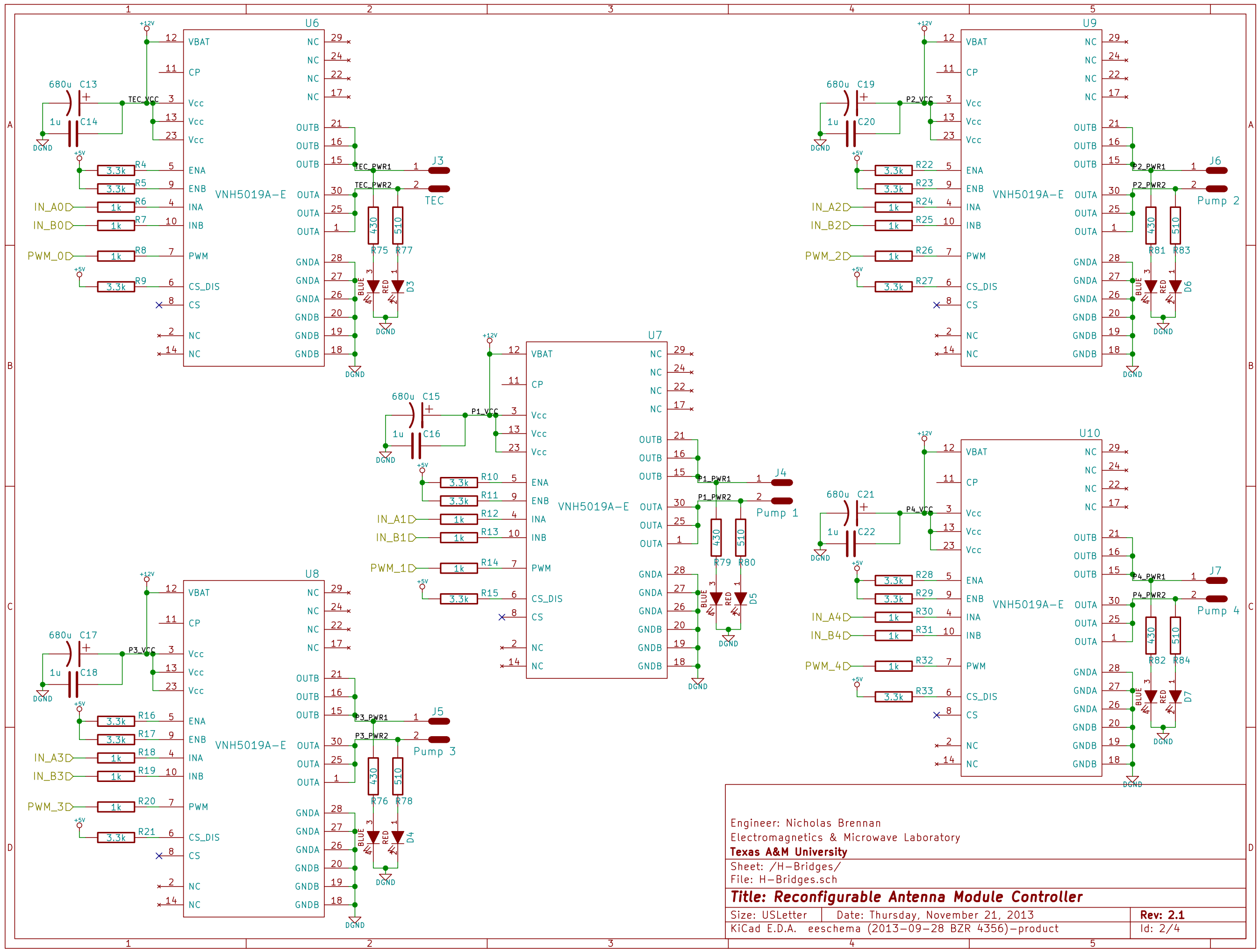


Fig. D.2: Modular control board: H-bridge circuitry

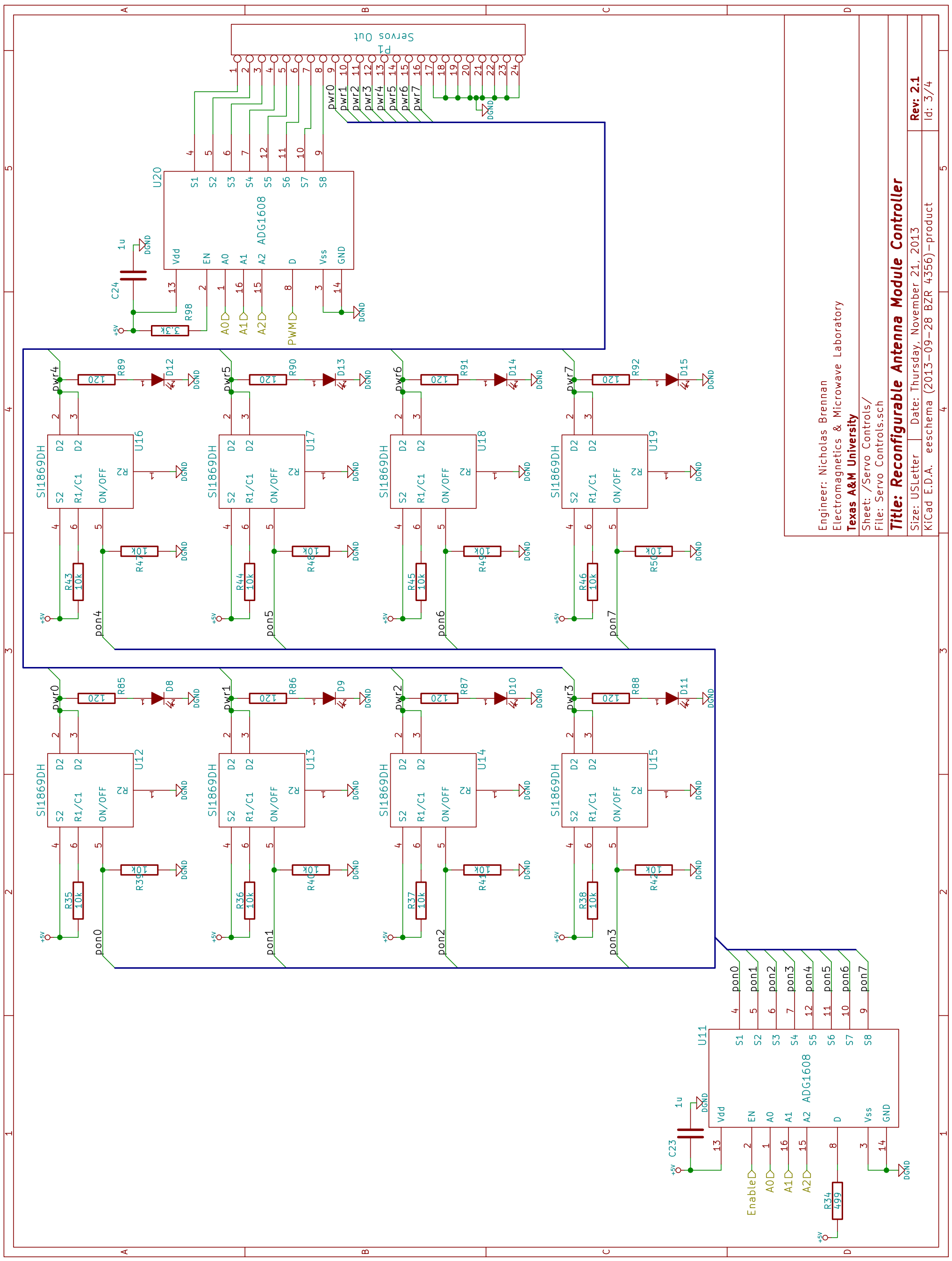


Fig. D.3: Modular control board: servo valve controls

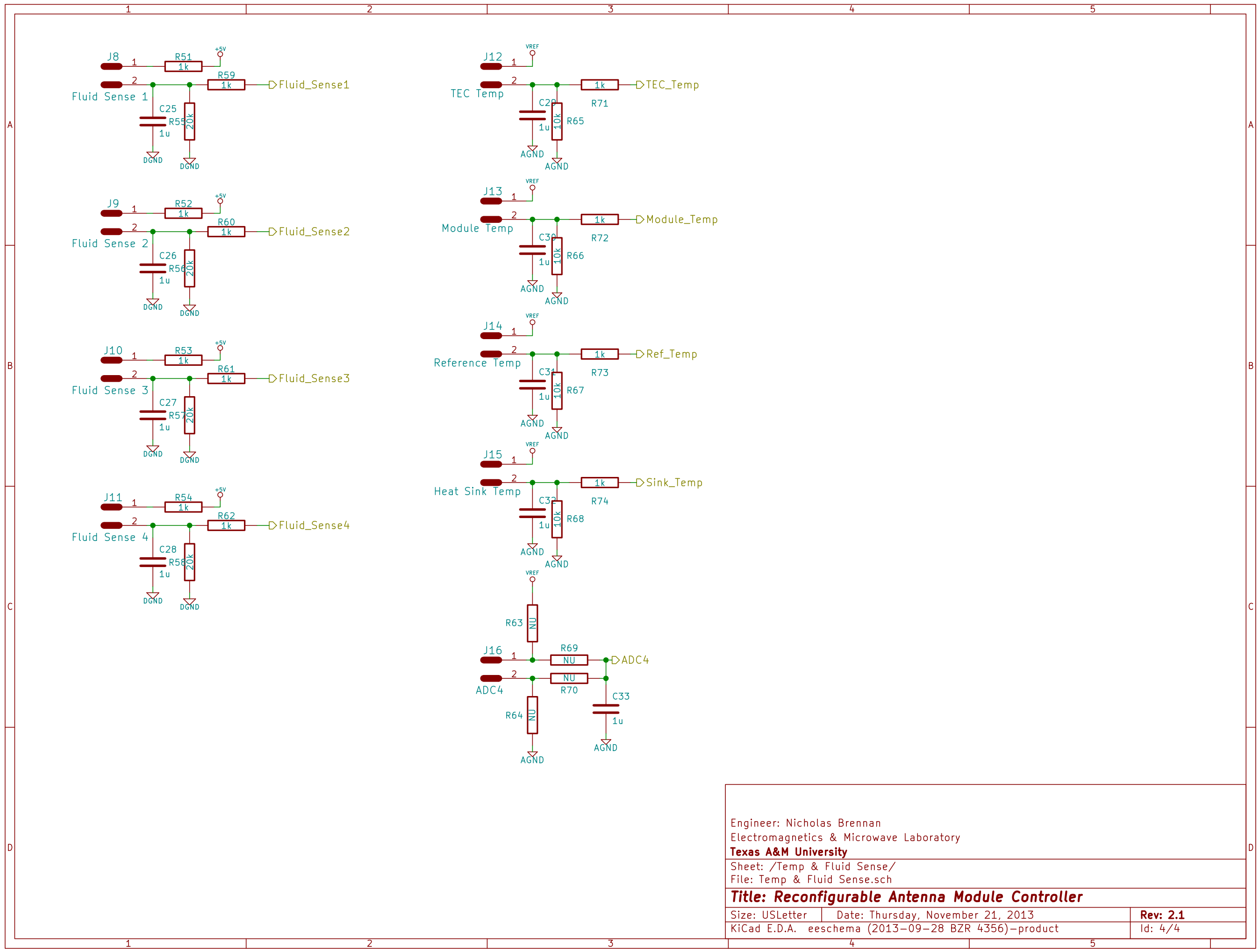


Fig. D.4: Modular control board: sensor inputs

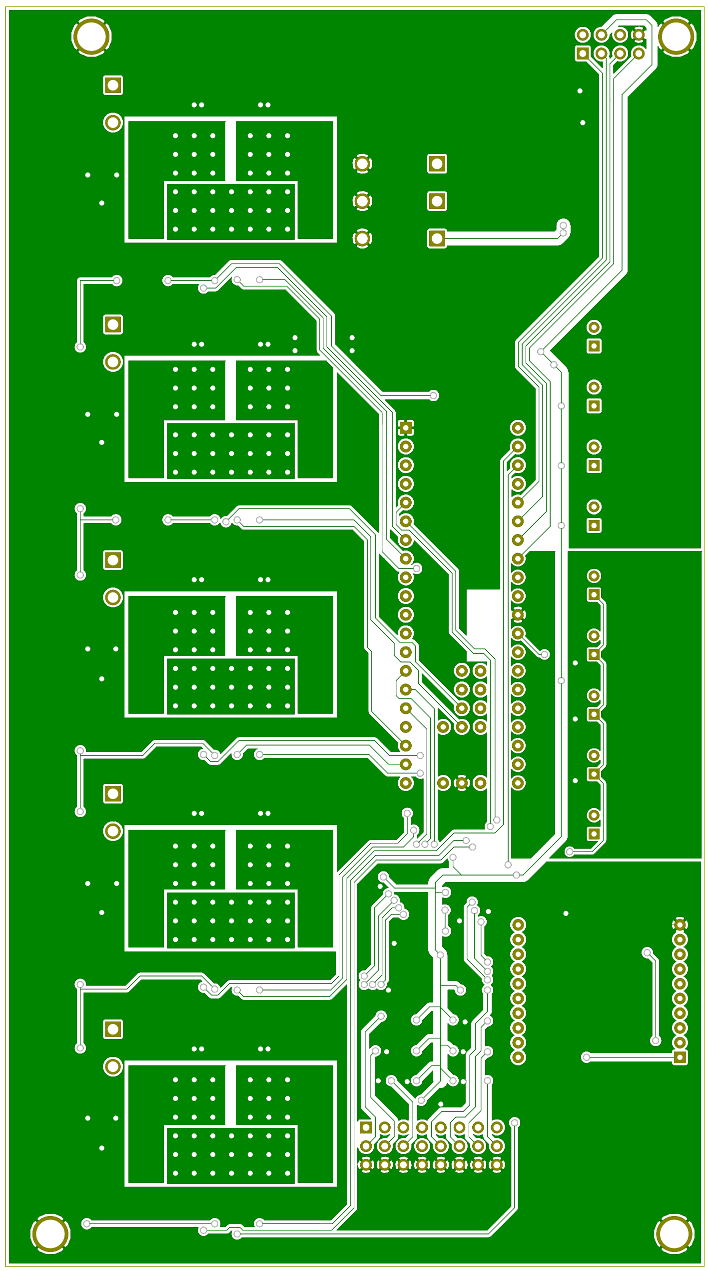


Fig. D.5: Modular control board PCB: bottom copper

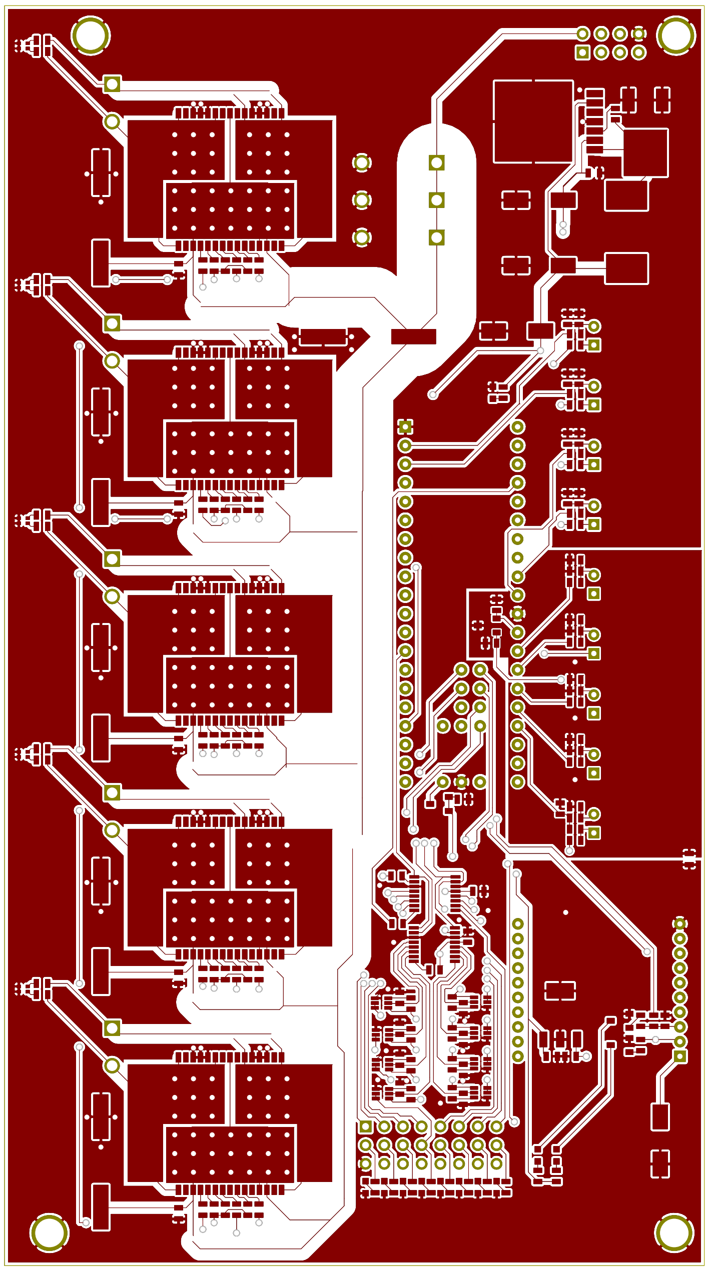


Fig. D.6: Modular control board PCB: top copper

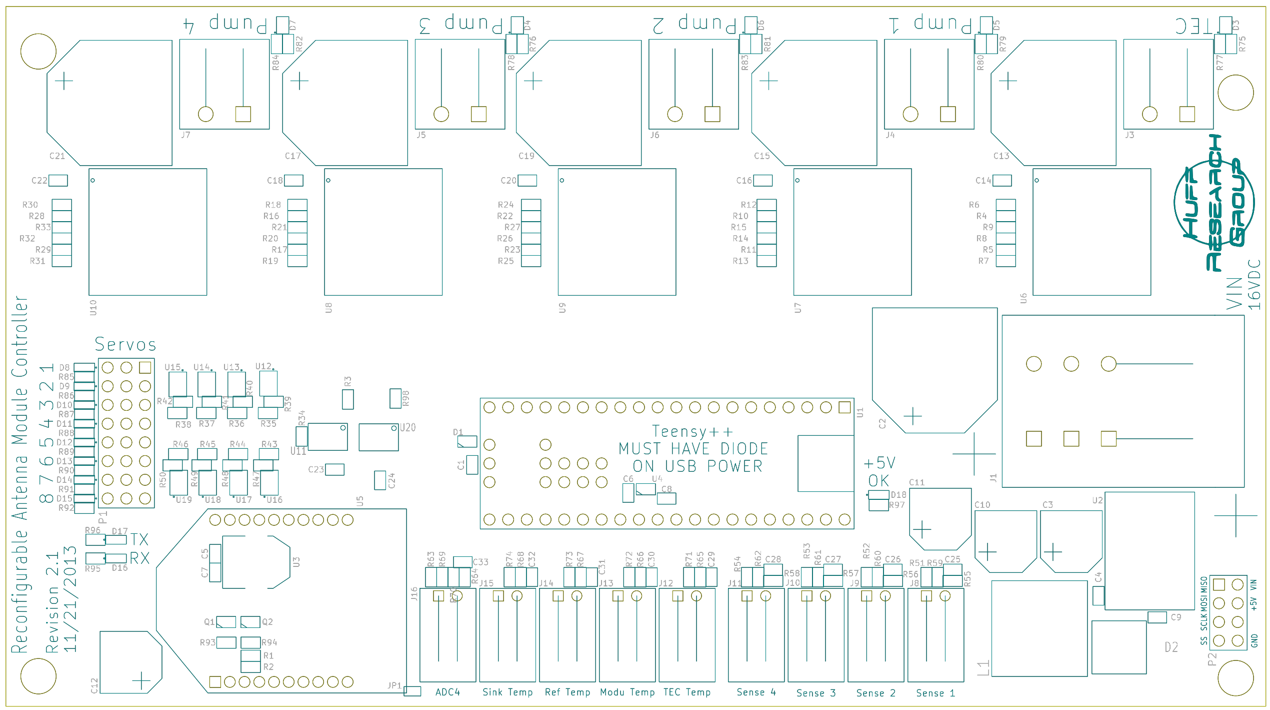


Fig. D.7: Modular control board PCB: silkscreen

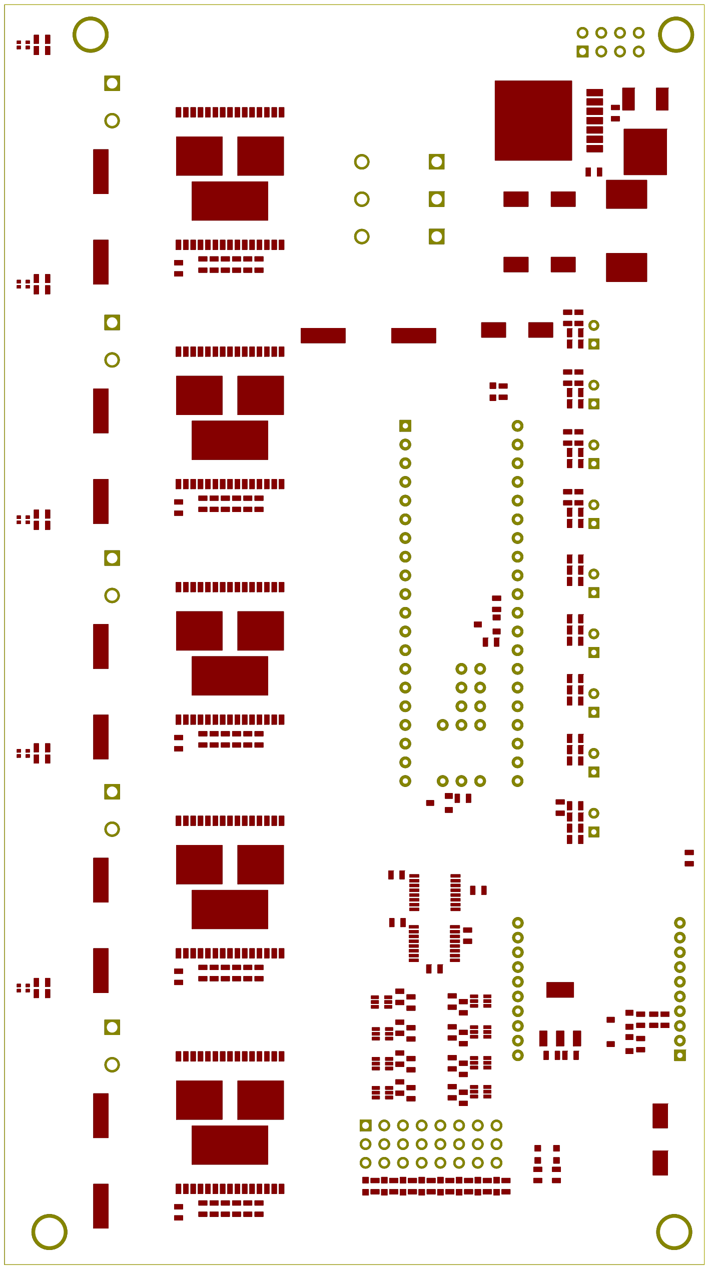


Fig. D.8: Modular control board PCB: soldermask