# An Automatic Control Interface for Network-Accessible Embedded Instruments

Graeme Smecher
Three-Speed Logic Inc., Vancouver
McGill University, Montreal
gsmecher@threespeedlogic.com

François Aubin
McGill University, Montreal

Elizabeth George
University of California, Berkeley

Tijmen de Haan
McGill University, Montreal

James Kennedy
McGill University, Montreal

Matt Dobbs
McGill University, Montreal

## ABSTRACT

We describe a metaprogrammed control interface and support library for network-accessible embedded systems. Together, this project permits functions written in standard C code to be exposed via a network interface expressed in JSON. In turn, this JSON interface mates with a Python library that provides a high-level, user-friendly, and expressive development environment.

This control interface removes the need to explicitly code interactions at the Python and network layers. As a result, the volume of error-prone and redundant hand-written code (e.g. for error-checking and validation) is vastly reduced.

## Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and Embedded Systems; C.4 [**Performance of Systems**]: Design Studies

## General Terms

Design, Performance

## Keywords

Metaprogramming, Remote Procedure Call (RPC), Embedded Linux, Field-Programmable Gate Arrays (FPGAs)

## 1. INTRODUCTION

The McGill Cosmology Instrumentation Laboratory has developed and supported readout electronics for mm-wavelength telescopes. Our readout system, the Digital Frequency-domain Multiplexer (dfmux), is actively being used in high-impact observational cosmology experiments including EBEX [Aubin et al. 2010], PolarBear [Lee et al. 2008], and at the South Pole Telescope [McMahon et al. 2009]. These instruments are making observations of the Cosmic Microwave Background (CMB) radiation with unprecedented precision, unlocking new information about the origin, growth, and ultimate fate of the universe.

The dfmux system biases and reads out bolometer arrays using Frequency Domain Multiplexing (fmux), a task requiring a large number of modulators and demodulators operating in parallel. The dfmux implements these in a Field-Programmable Gate Array (FPGA). In addition to biasing bolometers and reading out data, this FPGA is the primary control interface for setting up, monitoring, and maintaining sensors and associated electronics within the experiment.

The dfmux is well documented from physics and signal-processing perspectives [Dobbs et al. 2007, MacDermid et al. 2009, Smecher et al. 2012]. From a software perspective, we have shifted over the past few years from a largely hand-written mixture of Python and C code to a base of code that is largely self-constructing and self-documenting. This shift, using "metaprogramming" techniques, is intended to improve both code quality and developer efficiency. In this paper, we describe this evolution, and describe its impacts on the system's quality and performance.

The extraordinary amount of relevant software makes it difficult to provide a concise survey of related work. As a Remote Procedure Call (RPC) interface, this work is similar to JSON-RPC [JSON-RPC Working Group 2011]. As an embedded platform, it owes a great deal to the collaboration between Xilinx and PetaLogix, which has produced a solid technical basis and supportive community for running Linux running on Xilinx's FPGAs.

This paper is organized as follows: in Section 2, we present a brief description of the dfmux. (While we use the dfmux as a motivating example throughout this work, the same control interface is in use in our cryogenic fridge controller board.) Section 3 describes the firmware stack (where firmware is used loosely to refer to the FPGA bitstream, kernel, and application code that make up a complete software

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
Copyright 20XX ACM X-XXXXX-XX-X/XX/XX ...$10.00.

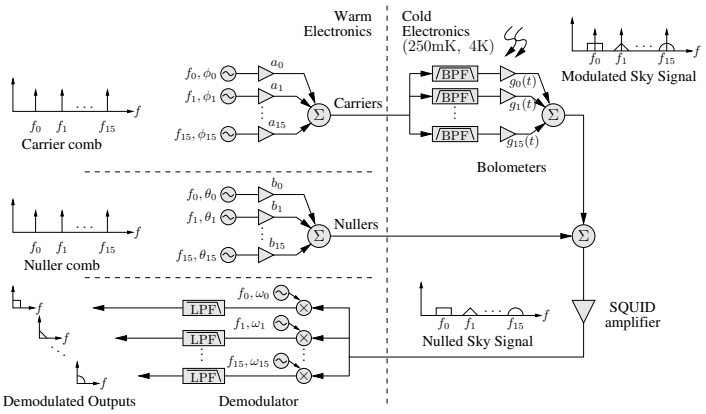


**Figure 1: Model of a frequency-multiplexed bolometer readout. 16 bolometers are multiplexed onto one set of wires crossing into the cryostat. A complete deployment uses many such modules.**

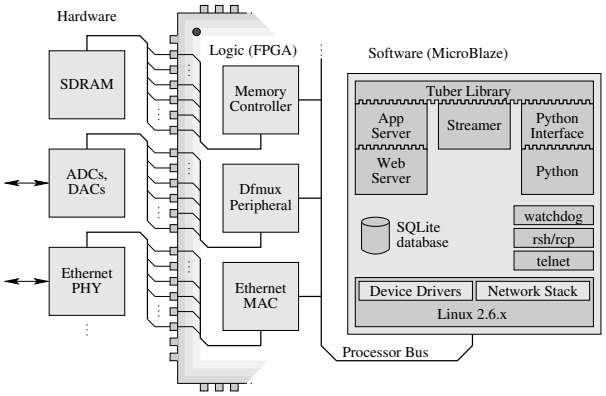


**Figure 2: The boundaries between hardware, FPGA logic, and software for the dfmux. Other peripherals/subsystems omitted for simplicity.**

release for the dfmux.) We remark on the system's performance in Section 4, and conclude with Section 5.

## 2. THE DFMUX SYSTEM

In millimeter-wave bolometer-array telescopes, the detectors (bolometers) operate at cryogenic temperatures. In such systems, the wires crossing between room-temperature and cryogenic portions of the system transmit heat conductively and place a significant load on the cooling system. For telescopes with hundreds or thousands of bolometers, some form of multiplexing (placing many detectors on a smaller number of wires) is necessary to keep the cooling system practical.

The dfmux multiplexes bolometers using frequency-domain multiplexing. Each bolometer in this system is tuned to an unique, narrow frequency band using an analog filter, and is biased by a carrier signal. As the image incident on the telescope's focal plane changes, each bolometer behaves as a time-varying gain, producing amplitude modulation (AM) sidebands around each carrier frequency. The dfmux digitizes and demodulates these signals, and streams the resulting signals across an Ethernet network.

Figure 1 shows a dfmux readout system. The dashed vertical line represents the barrier between warm and cold electronics; in this model, everything to the left of this line occurs within a Field-Programmable Gate Array (FPGA).

In addition to its real-time biasing and readout tasks, the dfmux performs a variety of control interactions that configure the system's frequencies, gains, phases and control switches; monitor the system's health; et cetera. These interactions occur via the dfmux's software stack, which is generally controlled using Python code typified by the following:

```
>>> import dfmux
>>> d = dfmux.Dfmux('192.168.0.72')
>>> d.set_mezz_power(1, power=True)
>>> d.set_fir_stage(3)
>>> d.set_frequency(frequency=690e3, units=d.HZ,
...         channel=1, target=d.DEMOD, module=1)
>>> d.launch_streamer('224.168.0.1', '8888')
```

**Listing 1: Interacting with the dfmux**

This Python snippet configures the dfmux to decode an AM radio station. It powers up a dfmux's analog mezzanine, which contain the ADCs and DACs. Then, it configures the output sampling rate, sets a demodulator frequency to 690 kHz, and begins multicasting data to the board's Ethernet port. With a suitably coupled antenna at the ADC input and a short program on a networked PC, these 6 lines of Python code would produce intelligible results.

In the following sections, we describe how this interaction takes place, beginning on the board and moving up the stack and across the network.

## 3. FIRMWARE/SOFTWARE STACK

Firmware development for the dfmux began in 2006, using the FPGA vendor's IP stack (lwIP) and lightweight kernel (xilkernel). Beginning in 2007, we evaluated and ultimately adopted an early version of Linux for the MicroBlaze, based on Linux 2.4. Then, beginning in 2010, we began a parallel development effort to replace the 2007-era software stack with a more modern version based on Linux 2.6. This updated stack was first released in 2011, and has since become the recommended firmware for dfmux deployments.

Figure 2 diagrams the firmware stack, showing the boundaries between hardware, logic, and software. This diagram focuses on the dfmux peripheral (which performs modulation and demodulation), the Ethernet interface (where streamed data are transmitted), and the external memory subsystem (shown for context.) Each of these three examples is supported by a combination of hardware, logic, and software.

### 3.1 Logic-to-Software Interface

The dfmux has several custom peripherals, the main example of which contains the synthesizer and demodulator logic for biasing and readout of bolometers. This peripheral synthesizes signals continuously, without ongoing interaction with the MicroBlaze. It also reads out data continuously, writing the results to a buffer that is formatted and flushed to the network via the software streamer.

Typical operations with this peripheral include setting or retrieving each synthesizer or demodulator channel's phase,

amplitude, or frequency setting. More complex interactions include aligning the sampling instants of multiple dfmux boards within a crate. Where it is feasible, these operations are performed directly from userspace to hardware by memory-mapping the peripheral directly [Corbet et al. 2005] and without involving oversight from the kernel. This results in a very short kernel module that implements the `mmap` system call. In user-space, hardware can be directly accessed as follows (where error-handling is omitted):

```
int device_fd;
struct dfmux_regs *mmap_regs;
device_fd = open("/dev/dfmux", O_RDWR);
mmap_regs = mmap(0, 4096, PROT_READ|PROT_WRITE,
    MAP_SHARED, device_fd, 0);
mmap_regs->base_control = 0;
```

**Listing 2: Direct Hardware I/O from Userspace**

The last line directly accesses hardware; the first lines are set-up and need not be called repeatedly. By avoiding kernel-mode drivers, context switches are minimized and device interaction is much faster. In addition, the amount of kernel-mode code to maintain drops greatly; if existing kernel frameworks such as Userspace I/O (UIO) are used, there may be no custom kernel code at all. Finally, code written in this manner can generally be upgraded by updating a user-space program, which can often be done on-line without rebooting the board.

For peripherals with more complex interfaces (e.g. emptying the readout buffer, which involves DMA and interrupts), we rely on traditional in-kernel device drivers.

## 3.2 C Interface Library

A single, coherent C library provides local access to the board's hardware. All of the board's interfaces (Python running locally or remotely, or the web-based interface, or local C programs such as the data streamer) follow the same code path through this library. Common tasks such as error handling are centralized, and are thus consistent rather than being duplicated in an ad-hoc basis for each command.

The higher-level APIs (e.g. Python) directly reflect the API exposed by this C library. We augment each C function with macros that permit the library to be self-describing. For example, the following code is expanded using the tuber_method macro to emit an ordinary C function, a second C wrapper that accepts and validates arguments in JavaScript Object Notation (JSON) [Crockford 2006] form, and a meta-data structure that describes the call's purpose, arguments, their defaults, and return type:

```
tuber_method(Dfmux, VOID, set_mezz_power,
    "Set mezzanine power.",
    2, (    /* 2 arguments, described below */
        (INTEGER, m, NULL, "Which mezzanine?"),
        (BOOLEAN, power, json_true(), "Power on?")
    ),
    "There are two mezzanines, indexed 1 and 2."
) {
    VALIDATE_MEZZANINE(m,); /* ensure m=1 || m=2 */
    if(power)
        self->mmap_regs->base_regs.control |= \
            m==1 ? 0x8000u : 0x80000000u;
    else
        self->mmap_regs->base_regs.control &= \
            m==1 ? ~0x8000u : ~0x80000000u;
}
```

**Listing 3: The set_mezz_power function**

This listing shows the only code (in Python or C) that specifically involves setting the mezzanine power. It is typically invoked using the JSON wrapper function, which accepts and returns arguments expressed as JSON objects using the open-source Jansson [Lehtinen 2012] library. This JSON wrapper can be called by C code, and includes automatically generated argument validation and error-handling code according to the function's arguments.

The code and data generated by the tuber_method macro are linked into a shared library, which is used from several contexts to invoke function calls.

In the following sections, we describe how both remote and local Python sessions interact with this library. Although we focus on Python code, the dfmux also provides a browser-based front-end that uses JavaScript to interact with hardware. Finally, some experiments maintain their own C/C++ code that interacts with the board over the network, using the same JSON interface.

## 3.3 Remote Python Adapter

The JSON interface described above is exposed across the network using a webserver and FastCGI [Payne 1996] backend, permitting Python and JavaScript code to easily interact with the board. For example, the dfmux would dispatch any of the following JSON objects to the set_mezz_power function shown above.

```
{ "object": "Dfmux", "method": "set_mezz_power",
    "args": [ 1, true ] }

{ "object": "Dfmux", "method": "set_mezz_power",
    "kwargs": { "m": 1, "power": true } }
```

**Listing 4: Simple JSON Method Calls**

To execute these calls, the Ethernet hardware must receive a packet and pass it to the Linux network stack, which passes it to the webserver, which passes it to the FastCGI backend. Once the JSON contained in the request has been decoded and executed, and the response encoded back into JSON, the same process repeats itself in reverse to send a response packet. Compared to the relatively trivial bit-flips involved in some of these calls, the overhead and latency of this process is onerous. To improve performance, we provide a "bulk" interface that permits a number of sequential calls to be combined into a single request and executed together:

```
[ { "object": "Dfmux", "method": "set_mezz_power",
    "args": [ 1 ] },
  { "object": "Dfmux", "method": "set_mezz_power",
    "args": [ 2 ], "kwargs": { "power": false } } ]
```

**Listing 5: A Bulk JSON Method Call**

This JSON snippet turns mezzanines 1 on, and 2 off. It demonstrates the use of default values (the "power" argument is missing in the first call) and mixed array/associative arguments in the second call. This argument-passing style closely matches Python's semantics.

Python sessions interact with this interface using an adapter class. This adapter uses Python's introspection capabil-

ities to intercept method calls and marshal them into JSON requests, which are invoked on the board. This Python adapter contains no code specific to a particular dfmux firmware release; instead, it queries the dfmux about what methods are supported using the metadata described above.

The automatic interface between Python code and network requests decouples Python from firmware upgrades, and removes the need to add boilerplate code in Python whenever C code is changed. It replaces a sprawling Python codebase with a very short piece of generic dispatch code.

In addition to marshalling calls, the Python adapter class includes several aids for interactive Python sessions e.g. using ipython [Pérez and Granger 2007]. These aids include tab-completion and self-documentation via DocStrings. These aids are implemented using the metadata stored by the tuber_method macro.

### 3.4 Local Python Adapter

For Python sessions running on the board, we have developed a Python module that provides direct access to hardware. This module uses the JSON methods exported by the tuber_method macro, directly exposing the Jansson objects used by these methods to Python code, without any serialization or deserialization.

On the Python side, introspection is again used to intercept method calls and convert them into requests for the tuber module. Since the on-board Python environment is rarely used interactively, no effort has been invested into tab-completion or DocString support.

## 4. PERFORMANCE

In this section, we remark on the system's performance. We focus on those design elements that are relevant for similar systems.

### 4.1 Heavy or Light?

A previous incarnation of this firmware operated without a Memory Management Unit (MMU). This led to stability problems due to memory fragmentation. After adding an MMU, we noted that the overhead associated with context switches rose substantially. This increase drove us to streamline device interactions, by (for example) replacing kernel-mode drivers with direct I/O from userspace. We have also worked to minimize interrupt overhead (by maximizing the amount of useful work accomplished by each interrupt).

After our C code is optimized to avoid context switching, performance is acceptable. However, the front-end (network stack, webserver, FastCGI interface, and JSON serialization/deserialization) still imposes a significant overhead compared to the function calls themselves, which are often as simple as setting or clearing a bit in a hardware register.

We recommend a careful evaluation of the framework that best suits each application. For our firmware stack, it was a logical necessity to add an MMU and accept the attendant decrease in performance. However, the MMU exemplifies our drift to a relatively heavyweight stack, which prioritizes flexibility and ease of development over simplicity and performance. Such a firmware stack is not always appropriate.

Heavyweight stacks also come with the risk of design marginalization, in which specific design elements (e.g. the CPU or toolchain) make it difficult to adopt the most suitable or widely adopted tools. For example, MicroBlaze does not yet support the standard multi-threading library for Linux [Drepper and Molnar 2003]. Any software that does not work on the older and more restrictive threading library available on MicroBlaze cannot easily be used on the dfmux.

### 4.2 API Unification

One of the main successes of our firmware renovation is the unification of the Python and C APIs. Not only has this made the Python API largely automatic, it has changed the process of writing C code on the board from a minefield to a relatively powerful, painless environment.

In addition to keeping C coders happy, this unification has improved code quality. Errors in on-board code used to be handled explicitly, at several levels (e.g. in a kernel driver, in the C library invoking it, in a CGI interface to the library, and finally, in the Python code calling the CGI interface.) Keeping error-checking code synchronized and complete across different firmware releases was a difficult and error-prone process. Now, not only is error-checking centralized in C code, it is largely automated. Bugs in the JSON interface, for example, are ultimately bugs in the headers that define the tuber_method macro, and are much more likely to be exercised and fixed than an inconsistency in a particular method.

### 4.3 Floating-Point

In a high-level API, it is difficult to avoid floating-point quantities. For example, when setting a channel's frequency in Python, the following calls are equivalent:

```
>>> d.set_frequency(690e3, units=d.HZ,
...         channel=1, target=d.DEMOD, module=1)
>>> d.set_frequency(118541097, units=d.RAW,
...         channel=1, target=d.DEMOD, module=1)
```

**Listing 6: Floating-Point Interaction**

In the first case, the call specifies "human" units (Hertz); the second case uses "machine" units (a 32-bit register value). In both cases, this value ends up as a double-precision value in C. (Double-precision floats can specify 32-bit signed or unsigned values without ambiguity.)

Such an API is powerful, since it provides both high- and low-level access to hardware through a single method. However, since the MicroBlaze does not have a hardware floating-point unit (FPU), any code involving these quantities is compiled into slow floating-point emulation code.

We raise two important limitations with this use of floating-point emulation. First, casual use of floating-point emulation is only acceptable for calculations that are rare. It is not appropriate (and causes a substantial slowdown) when repeated in a loop, e.g. when retrieving and serializing samples from the streamer.

Secondly, although a single-precision FPU is available for MicroBlaze, commonly available C code (and large parts of the C runtime and language) relies on double-precision floating-point and cannot benefit from a single-precision FPU. For example, the Jansson JSON library relies on the scanf/printf family of functions for string serialization and deserialization. These functions are variadic (i.e. accepting a variable number of arguments). However, the C language specifies that only double-precision floating-point quantities are passed to variadic functions; single-precision floats are up-casted. Probably for this reason, printf/scanf-type functions do not explicitly handle single-precision values, and

any function calling them will have to contend with double-precision arithmetic even when dealing with single-precision data. Overall, single-precision FPUs with emulated double-precision math is a relatively common scenario that also applies to many ARM CPUs.

We are currently using floating-point emulation with caution, and accepting the performance impact. The CPU and FPGA industries seem to be gradually increasing the availability of double-precision FPUs, so this issue may become less important in the future.

# 5. CONCLUSIONS

In this paper, we described the control logic, firmware, and software associated with McGill's dfmux readout system. We described the control interface, which generates a JSON interface layer to C code, and exports this JSON interface to Python sessions running both on- and off-board. The control interface uses metaprogramming techniques to automatically generate validation and interface code. Compared to earlier iterations of the dfmux's software stack, this approach has greatly reduced the amount of code written and maintained by hand.

# 6. ACKNOWLEDGMENTS

The McGill authors acknowledge funding from the Natural Sciences and Engineering Research Council, Canadian Institute for Advanced Research, and Canadian Foundation for Innovation. MD acknowledges support from an Alfred P. Sloan Research Fellowship and Canada Research Chairs program.

# 7. REFERENCES

[Aubin et al. 2010] Aubin, F., Aboobaker, A. M., Ade, P., Baccigalupi, C., Bao, C., Borrill, J., Cantalupo, C., Chapman, D., Didier, J., Dobbs, M., Grainger, W., Hanany, S., Hubmayr, J., Hyland, P., Hillbrand, S., Jaffe, A., Johnson, B., Jones, T., Kisner, T., Klein, J., Korotkov, A., Leach, S., Lee, A., Limon, M., MacDermid, K., Matsumura, T., Meng, X., Miller, A., Milligan, M., Polsgrove, D., Ponthieu, N., Raach, K., Reichborn-Kjennerud, B., Sagiv, I., Smecher, G., Tran, H., Tucker, G. S., Vinokurov, Y., Yadav, A., Zaldarriaga, M., and Zilic, K. 2010. First implementation of TES bolometer arrays with SQUID-based multiplexed readout on a balloon-borne platform. In *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*. Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series Series, vol. 7741.

[Corbet et al. 2005] Corbet, J., Rubini, A., and Kroah-Hartman, G. 2005. *Linux Device Drivers* 3rd Ed. O'Reilly.

[Crockford 2006] Crockford, D. 2006. The application/json media type for JavaScript Object Notation (JSON). http://www.ietf.org/rfc/rfc4627.txt.

[Dobbs et al. 2007] Dobbs, M., Bissonnette, E., and Spieler, H. 2007. Digital frequency domain multiplexer for mm-wavelength telescopes. In *Proc. 15th IEEE-NPSS Real-Time Conference*. 1–6.

[Drepper and Molnar 2003] Drepper, U. and Molnar, I. 2003. The native POSIX thread library for Linux. Tech. rep., Red Hat, Inc.

[JSON-RPC Working Group 2011] JSON-RPC Working Group. 2011. JSON-RPC 2.0 specification. http://jsonrpc.org/specification.

[Lee et al. 2008] Lee, A. T., Tran, H., Ade, P., Arnold, K., Borrill, J., Dobbs, M. A., Errard, J., Halverson, N., Holzapfel, W. L., Howard, J., Jaffe, A., Keating, B., Kermish, Z., Linder, E., Miller, N., Myers, M., Niarchou, A., Paar, H., Reichardt, C., Spieler, H., Steinbach, B., Stompor, R., Tucker, C., Quealy, E., Richards, P. L., and Zahn, O. 2008. POLARBEAR: Ultra-high Energy Physics with Measurements of CMB Polarization. In *American Institute of Physics Conference Series*. American Institute of Physics Conference Series Series, vol. 1040. 66–77.

[Lehtinen 2012] Lehtinen, P. 2012. Jansson documentation. http://www.digip.org/jansson/doc/2.3/.

[MacDermid et al. 2009] MacDermid, K., Hyland, P., Aubin, F., Bissonnette, E., Dobbs, M., Hubmayr, J., Smecher, G., and Warraich, S. 2009. Tuning of kilopixel transition edge sensor bolometer arrays with a digital frequency multiplexed readout system. In *AIP Conf. Proc.* Vol. 1185. 253–256.

[McMahon et al. 2009] McMahon, J. J., Aird, K. A., Benson, B. A., Bleem, L. E., Britton, J., Carlstrom, J. E., Chang, C. L., Cho, H. S., de Haan, T., Crawford, T. M., Crites, A. T., Datesman, A., Dobbs, M. A., Everett, W., Halverson, N. W., Holder, G. P., Holzapfel, W. L., Hrubes, D., Irwin, K. D., Joy, M., Keisler, R., Lanting, T. M., Lee, A. T., Leitch, E. M., Loehr, A., Lueker, M., Mehl, J., Meyer, S. S., Mohr, J. J., Montroy, T. E., Niemack, M. D., Ngeow, C. C., Novosad, V., Padin, S., Plagge, T., Pryke, C., Reichardt, C., Ruhl, J. E., Schaffer, K. K., Shaw, L., Shirokoff, E., Spieler, H. G., Stadler, B., Stark, A. A., Staniszewski, Z., Vanderlinde, K., Vieira, J. D., Wang, G., Williamson, R., Yefremenko, V., Yoon, K. W., Zhan, O., and Zenteno, A. 2009. Sptpol: an instrument for CMB polarization. *AIP Conference Proceedings 1185*, 511–514.

[Payne 1996] Payne, D. 1996. FastCGI: A high-performance web server interface. http://fastcgi.com.

[Pérez and Granger 2007] Pérez, F. and Granger, B. E. 2007. IPython: a System for Interactive Scientific Computing. *Comput. Sci. Eng. 9,* 3, 21–29.

[Smecher et al. 2012] Smecher, G., Aubin, F., Bissonnette, E., Dobbs, M., Hyland, P., and MacDermid, K. 2012. A biasing and demodulation system for kilopixel TES bolometer arrays. *IEEE Trans. Instrum. Meas 61*, 251–260.