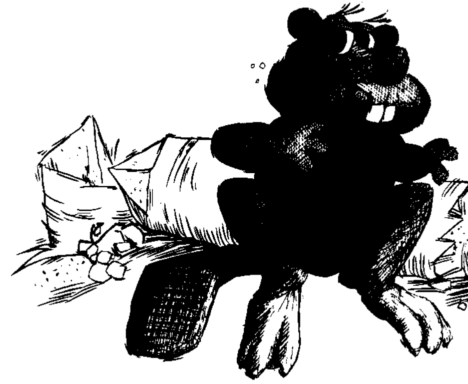
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT



A Reliable Typesetting System for Waterloo

Joseph H. Buccino

CS-80-20

April, 1980

A Reliable Typesetting System for Waterloo

Joseph H. Buccino

ABSTRACT

The Math Faculty Computing Facility at the University of Waterloo operates a Photon 737 Econosetter, a low speed photographic typesetting device interfaced to the Honeywell 66/60 timesharing system. In the years since it was purchased a number of people have worked on the Photon's hardware and software. Unfortunately very little of this work has been documented. The purpose of this essay is to provide documentation for the Photon system, and to describe the characteristics of each component in the system as completely as possible. A new software system has been developed to ensure that the typesetting process is reliable. It handles the problems encountered as data is passed from the operating system to the Photon. This essay explains not only what the software does, but also how it was developed and how it can be maintained. Source code, regeneration procedures and many useful techniques and procedures previously undocumented are included in the appendices as a guide to future users.

A RELIABLE TYPESETTING SYSTEM FOR WATERLOO

Table of Contents

1. Introduction
- 1.1 Preliminaries 1
- 1.2 Computing Environment 1
- 1.3 General Description 2

2. The Photon Econosetter
- 2.1 General Description 3
- 2.2 The Photo Unit 3
- 2.3 The Microdata 4

3. The Interface
- 3.1 Design Goals 6
- 3.2 Logic Description 6
- 3.3 The Interface Buffer 8
- 3.4 Communication 8
- 3.5 A Caveat 9

4. The Software System
- 4.1 Some Terminology 11
- 4.2 The Formatters (PROFF, TROFF and TYPE) 11
- 4.3 UNIX programs and Procedures 13
- 4.4 Some Problems on the Honeywell 14
- 4.5 Preparing a Program for the High-Core Loader 14
- 4.6 The Boot Procedure 15
- 4.7 The Pass Procedure 16
- 4.8 Alternative Solutions 17
- 4.9 Summary 17

5. Acknowledgements

6. References

Appendix A: Explain Files

Appendix B: Regeneration of the Software System

Appendix C: TROFF Output

Appendix D: Source Code

Appendix E: The Microdata

1. Introduction

1.1. Preliminaries

Typesetting of documents is provided to the users of the Math Faculty Computing Facility (MFCF) timesharing system via the Photon Econosetter. The general procedure is as follows. A user prepares a text file as input to one (or more) of the existing text formatters and preprocessors (TROFF, TBL, EQN, PROFF, TYPE) which produce a file of typesetting codes. This output is then passed to the Photon which interprets the codes as commands selecting the character, font and spacing, then exposes the final document on photographic paper. A built-in minicomputer controls the typesetting hardware within the Photon. But this is just an overview. There are a number of other steps which must be performed for all of this to run smoothly.

The Photon must be capable of interpretting the file passed to it, which requires loading the proper software into the minicomputer's memory. This process, known as 'booting' the Photon, is one of the two major problems addressed by this essay. The fact that more than one system is run on the Photon, and that there is no reliable way for a user to determine which software package is currently running, means that individual users must be able to load the proper software for their own purposes whenever they desire. This is a major difference from a standard typesetting shop where only one system is used, loaded once and never changed.

When it was acquired in the summer of 1974, the Photon was designed for this one-system operation, but the photocomposition capability was not the only reason for the purchase. In addition to being able to produce high quality text documents it was intended to provide valuable hands-on experience for faculty and students doing research on document preparation. The major work that has been done is the construction and programming of an interface that removes the typesetter's dependencies on its paper tape reader, and the programming of the Photon's internal minicomputer to accept more than the standard TTS typesetting language. Both of these projects will be explained in later sections of this essay.

As the capabilities of the typesetting hardware and software grew, so did its usage, particularly by users who were not interested in how typesetting was accomplished. These new users prefered to consider the Photon as a 'black box' that they could use to produce high quality output. Consequently, reliable and robust software was needed to meet this demand. This essay documents the development of the package which is currently being used for typesetting.

1.2. Computing Environment

To those unfamiliar with the computing facilities at the University of Waterloo a brief introduction follows. The timesharing system provided by the Math Faculty Computing Facility, TSS, runs on a Honeywell 66/60 mainframe. A Datanet 355 acts as a front-end, communicating with user terminals. TSS provides a wide variety of general and specialized timesharing services as well as access to batch processing. The other system that was used in developing the new software is the UNIX operating system which runs on a PDP-11/45. Programs and data can be sent between Honeywell and UNIX (without the need for an

intermediate storage device) using locally defined commands to the respective systems. Data is transferred by a set of programs known as the 'UNIX Daemon'.

1.3. General Description

The Photon acts as a receive-only device and the Honeywell system considers the Photon to be a terminal, just like all the others. The Photon considers the operating system to be a paper tape. Because there is no way for the Photon itself to send information to the system, it shares its communication line with a terminal. A locally-built interface controls the data-flow between the Honeywell, the terminal and the Photon by switching the line between the two output devices. The line is switched by interpretting certain transmitted codes as switching commands. Data is sent to either the typesetter or the terminal depending upon the state of the interface. This type of control over a communication line is made possible by installing the interface between the operating system and the two devices, allowing it to shut off communication with either of them.

The software which controls the Photon and the interface runs on the Honeywell 66/60 under TSS and is mostly written in the programming language B. It was necessary to deviate from this environment for some stages of the development since the software that runs on the Photon's minicomputer was developed on the UNIX system.

To avoid repetitive descriptions we first make a few definitions for referring to the different hardware components. When we say 'Honeywell' we mean the operating system, the Datanet and the mainframe. The word 'UNIX' will be used similarily to denote the operating system and the PDP-11/45 hardware it runs on. 'Photon' will be used to refer to the entire typesetter, including its minicomputer. When we wish to refer to the minicomputer by itself, we will use the term 'Microdata', the manufacturer of the unit. The unit that acts as the switch, allowing us to control both the Photon and a terminal, will be called the 'interface' as above.

Understanding the design decisions which were made while developing the new Photon software package requires a more complete knowledge of the various components of this system. For this reason functional descriptions are given for the various components. We will consider the Photon typesetter first, as the other components depend heavily upon its characteristics. This will be followed by a description of the interface and then of the software.

2. The Photon Econosetter

2.1. General Description

The Photon 737 Econosetter is an electro-mechanical phototypesetter with a self-contained Microdata minicomputer which controls the typesetting components. The typesetter is connected to and controllable by the MFCF Honeywell 66/60 computer. The Photon was originally a paper tape machine, equipped with a high speed paper tape reader mounted on the cabinet. This device has since been removed because it became unrepairable and because a locally built interface to TSS made the reader obsolete. A complete description of the Econosetter is divided into two parts, a description of the photo unit and a description of the internal minicomputer – the Microdata. We will start with the description of the photo unit.

2.2. The Photo Unit

Typesetting is accomplished by generating a flash of light which travels through a master image of the symbol to be produced. The light is then reflected 180° by a mirror mounted on a moveable carriage which selects the horizontal placement of the character on the current line. Finally this reflected light travels through a lens which magnifies the character to the chosen point size, after which the light exposes photographic material, producing an image. The film can be moved vertically, either manually or automatically, by a series of rollers and guides. After the document has been typeset the exposed film is collected in a removable cassette by advancing the paper and using a built in paper cutter to free it from the rest of the roll. Film can then be developed in the standard manner. Figure 2.2.1 shows a pictorial layout of the components and indicates the optical path.

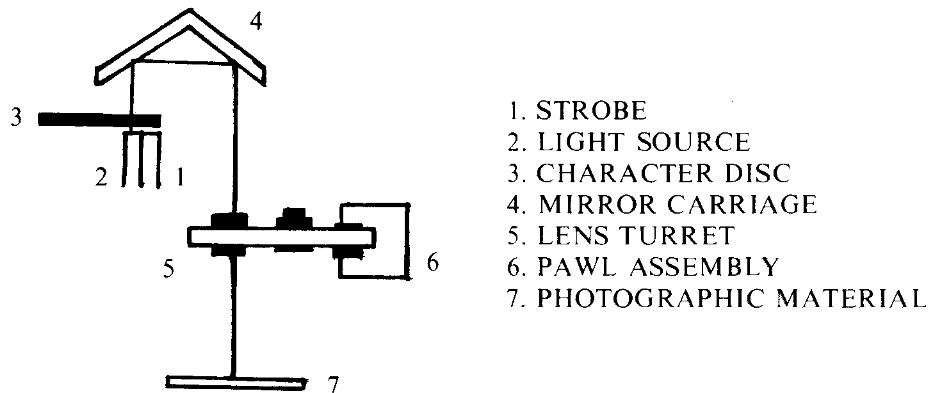


Figure 2.2.1

The flash unit consists of a high-voltage xenon flash component, appropriate electronic circuits and a power supply which produces a high intensity flash at the proper time to illuminate the desired character from a spinning disc. The master images of the four available fonts (Times Roman, Times Italic, Times Bold and Mathematics) are etched onto a glass disc covered by an emulsion. This unit is called the 'character disc'. Each font has 112 characters, as shown in Appendix C. The disc is belt driven and has timing marks which makes it possible for the flash to occur at the desired character.

The mirror carriage can move left and right along a track, positioning the light beam horizontally on the paper. Limit switches prevent the carriage motor from moving the carriage farther than the axis limits, although no software mechanism for utilizing these switches has been found by the author.

The final component in the system is a lens turret. This is a moveable wheel with lenses mounted in it, one for each point size. Our Photon has four point sizes: 8, 10, 14 and 18. (A point is 1/72 of an inch). The point size is changed by releasing a pawl on the turret so the wheel is allowed to rotate, then re-engaging the pawl to lock it in position when the turret has moved to the chosen lens position. As the wheel rotates, the Microdata can read its angular position, enabling it to decide when to stop the motion.

The photographic paper is an 8 inch wide, 150 foot continuous roll in a light-tight cardboard box. The hardware is able to sense when the unit is out of film and will cause the Microdata to halt and await a new box.

2.3. The Microdata

The Microdata is a sixteen bit, two's complement minicomputer with a twelve bit accumulator. It has no other general-purpose, stack or index registers. Instructions are sixteen bits and directly address all 4K of the sixteen bit memory words. There is also the capability to directly access each of the 8K eight-bit bytes. Core memory is exactly the size of this address space, 4K sixteen bit words. It retains its contents even when the Photon is powered down. Memory words are usually represented as four hexadecimal digits. Thus, the legal address range is 0000 - 0FFF.

The controls provided on the front panel of the Microdata include four SENSE switches, a RESET and RESTART switch, sixteen toggles and corresponding indicator LED's, and switches that allow the toggles to be used to enter a program into core. There is also a switch that causes the contents of the accumulator to be displayed on the indicator lights. Figure 2.3.1 shows the front panel of the Microdata. Use of the various switches is described in Appendix E.

One front panel switch that is used frequently is the LOAD switch. When depressed it will execute the hardware bootstrap loader. Originally this caused successive frames to be read from the paper tape reader (ignoring leading blank frames). Since the paper tape reader has been removed, the loader now reads successive bytes from the interface buffer (ignoring the ASCII character NUL – octal 000). The bytes read are stored into successive locations in core starting at the address given by the data switches. The most significant eight bits of a two-byte word are always loaded first, followed immediately by the least significant eight bits. The loader will display the address into which it is storing on the

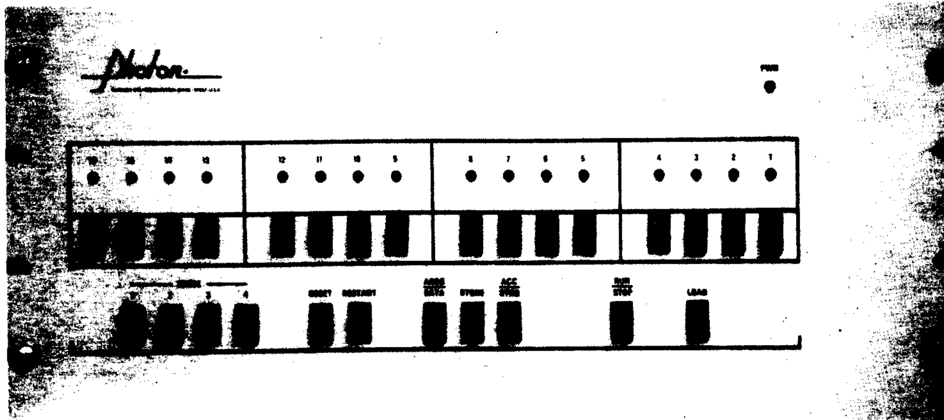


Figure 2.3.1

indicator lights and will eventually wrap around and start storing in location 0000 once it passes address 0FFF.

The instruction set is a very simple one. There is no provision for indexing, indirect addressing or stack manipulation. These constructs must be simulated in software. The set of available instructions is logically separated into two categories: the general use instructions and the I/O instructions. General use instructions all have the same format: a four-bit opcode and a twelve-bit operand or address. I/O instructions also share this format, but very few of the operands are documented by Photon. Appendix E lists all known instructions and their effects.

The interrupt mechanism provided is very elementary. Despite this, it is still not fully understood even now. When interrupts are turned on, the Microdata is interrupted every millisecond. When this interrupt occurs control is passed to location 0001 which is supposed to contain a subroutine jump instruction to the appropriate interrupt routine. Control of the individual components of the typesetting hardware uses this as a timing mechanism, as described below.

Most components of the photo unit do not set any internal flags when they are free or when their operation is finished. Instead they are documented to take a certain number of milliseconds to complete their operation. This is usually handled by initializing a counter when the operation is started and decrementing the counter in the interrupt routine. When the counter reaches zero the operation is assumed to be complete.

As mentioned above, this procedure is not fully understood. The difficulty lies in the undocumented use of some I/O instructions in the software provided by Photon; these apparently set up a return address from interrupts. Our solution is to consider these instructions as a requirement of any interrupt routine, and we blindly include them as they originally appeared in the supplied programs. Interrupt routines are included in both of the programs listed in Appendix D, so future programmers can copy them, adding or deleting counters where necessary.

3. The Interface

3.1. Design Goals

When the Photon was purchased its only input medium was paper tape. To typeset a document a user would first have to load the supplied software for the Microdata by paper tape. Since there was only one such program provided by Photon, and because the core retains its contents when powered down, this procedure did not have to be performed very often.

After a user had run his source text through the software that produced a file of bytes understandable by the typesetter, the file would then have to be transferred to paper tape before it could be typeset. In addition to being time-consuming and wasteful, this extra step proved even more difficult due to the unreliability of hardware to punch paper tape. It was obvious that it would be preferable to provide a direct connection between the Honeywell and the Photon so that typesetting codes could be transmitted in a straightforward manner. This was the motivation for the construction of the interface.

Ideally the interface was expected to behave identically to a paper tape reader, performing similar functions when the Microdata executed paper tape reader instructions. In this way the Microdata program for interpreting codes did not need to change. It was decided that the transmission of data be done by the Honeywell under user program control, as if the Photon were a terminal. This method has the advantage that the data can be stored in the file system just like any other data.

In summary, the design of the interface was to be so that it could be installed on the line between the operating system and the terminal allowing it to direct the flow of bytes to either the terminal or the Photon. Also, this switching should be done automatically so that users with no knowledge of the design of the system could typeset documents. Within these constraints the Computer Communications Network Group (CCNG) at the University of Waterloo designed and built an interface. What follows is a functional description of that unit as it now exists.

3.2. Logic Description

The interface is contained on an 8 × 9 inch circuit board installed in the base of the Photon's housing. There are two components of this unit: the logic which controls whether bytes being transmitted are to go to the terminal or to the Photon, and the buffer which collects bytes that will eventually be read by the Microdata. Outlets for lines to the terminal and the operating system are marked on the board. Figure 3.2.1 shows a schematic of how the interface is connected to the various components. The arrows indicate how data flows through the system.

The interface is automatically powered up when the Photon is turned on. It is also reset so that data is initially communicated between the terminal and the operating system. The operation of the interface's logic is very simple. It observes all transmission to and from the terminal until the byte octal 352 is transmitted in either direction. This byte will be called the Start Of Message signal or SOM*. From that point on all bytes go to the interface buffer (instead of the terminal) until the byte octal 331 is sent. This byte will be called EOM*

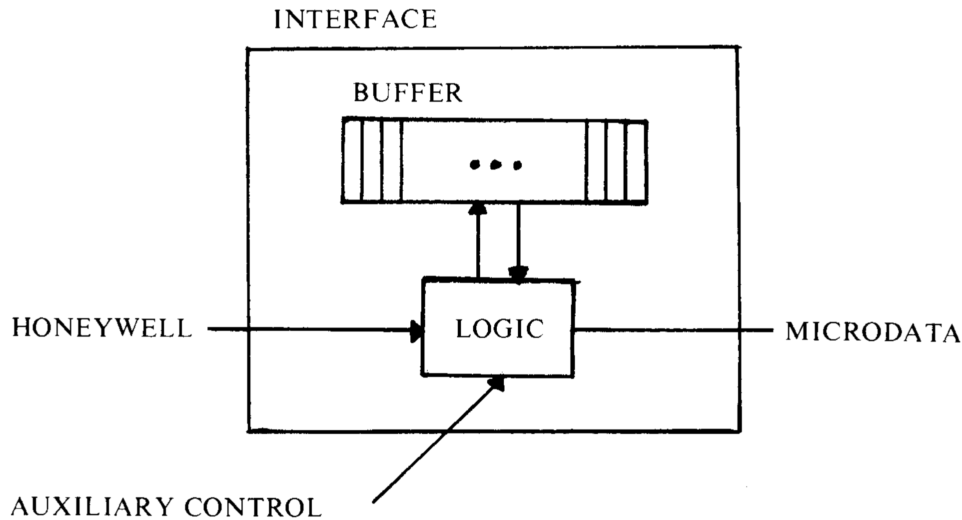


Figure 3.2.1

for End Of Message. Neither of these two bytes go into the interface buffer, but are used only for switching the line. A small letter 'j' appears on the terminal's screen when the SOM byte is sent. This is because the line is still directed to the terminal until after this byte is read. (Note that SOM is also the ASCII code for the character 'j' if the parity bit is ignored.) Similarly nothing can appear on the terminal until after the EOM signal is sent. EOM would print as a 'Y' if it could get through the interface logic to the terminal.

Choosing these switching codes from the printable ASCII character set would normally be disastrous, as an unaware user could accidentally switch the line. Even more, the bytes that comprise typesetting commands, interpretted by the Microdata, can contain these codes. It will be shown in the software section that these codes need not appear in any transmission to the typesetter. But even though this is the case we still have the undesirable situation that the letters 'j' and 'Y' cannot be sent to a terminal without unwanted side effects. This problem is solved in the following manner. Recall that ASCII codes are seven bits and that the Honeywell timesharing system uses even parity. Note also that both EOM and SOM as eight-bit codes have an odd number of bits set. The key idea is to operate in no-parity mode (which causes the Honeywell not to check and supply the parity bits) when sending a file to be typeset. The terminal's parity switch need never be touched because only the typesetting software should be turning the line so that the transmission goes into the interface buffer. In this way we preserve all the capabilities of a normal terminal as well as providing for typesetting.

There is only one other byte that is special to the interface; it will never go

---

* These abbreviations are not standard ASCII mnemonics.

into the buffer for the Microdata to read. This is the EOL* byte (End Of Line) – octal 021. This is the ASCII character DC1. Its purpose is to inform the interface (or terminal) that the system is waiting for a response. The reason this byte is ignored is that it is sent by the Datanet automatically after a group of bytes have been transmitted over the line. This is the standard way that most operating systems implement a hand-shaking mechanism with user terminals. It will become clear why this is necessary in the explanation that follows concerning the communication protocol with the interface.

3.3. The Interface Buffer

The buffer contains 256 eight-bit bytes arranged in a logically circular fashion with sufficient electronics to allow the Microdata access to these bytes. There are two independent pointers, one used to store bytes into the buffer, the other for reading bytes from the buffer.

The store pointer always indicates the next available byte for storage of data coming from the operating system. There is always an available byte for this store operation, even if the buffer is 'full'; the logic never checks to see if the buffer is full and will overwrite data if no software check is made. This is unfortunate, but cannot easily be changed, so the programmer has to make sure that the buffer never overflows.

The read pointer always indicates the first byte not yet read by the Microdata. The logic is designed so that the read pointer can never overtake the store pointer. The buffer is defined as empty when both the read and store pointers point to the same byte. It is possible to clear the buffer manually by hitting the 'break' key on the terminal. This is used in the protocol to ensure that the buffer does not have anything left in it from previous use.

If the read pointer could overtake the store pointer an empty buffer situation would be disasterous. Not only would the Microdata be reading bytes that have already been processed, but when the Honeywell finally sends the next buffer the read pointer has gone far enough so that some or all of the new bytes have been passed. This is because new bytes are added starting at the store pointer which can get arbitrarily far behind the read pointer under these circumstances. This would cause some bytes to be processed twice while other bytes would not be processed at all. Therefore, the interface was designed so the that the read pointer could not overtake the store pointer.

3.4. Communication

When the interface logic sees the EOL byte (meaning the system is waiting for a response) it automatically responds with a carriage return (octal 012) as soon as more than 128 positions are available in the buffer. This allows for synchronization so that the problem of buffer overflow can be avoided.

The Honeywell software sends bytes in groups of less than or equal to 129. Since each of these groups is automatically followed by an EOL, when the buffer responds with a carriage return the operating system can be assured that there is room for the next group of bytes. Clearly we cannot send more than 129 bytes at a time, because the EOL response indicates only that 129 byte positions are available. We have no assurance that more will be available when we are ready to

send them. This convention assures that TSS never gets far enough ahead of the Photon to overflow the buffer.

The complementary problem is far more common, and potentially more difficult to deal with. As the TSS load increases, the system cannot supply the interface buffer with bytes as quickly as the Microdata needs them. This can happen when the timesharing system has a large number of users, a quite common occurrence. When this happens the buffer will be empty, meaning that both the read and store pointers indicate the same position. As the interface was originally designed, if the Microdata attempted to read from the buffer in this state, the data returned was the same as the last byte read. Since the read pointer can never overtake the store pointer all subsequent requests from the Microdata for data would also return this same byte until more data is stored in the buffer. This effectively made the interface unusable during peak load times.

For this reason, the interface has been modified to partially correct this problem. When the interface buffer is empty a read issued by the Microdata returns an ASCII NUL. Since the Photon is a paper tape oriented machine all its software ignores these bytes (which are tape feeds on a paper tape) and issues another read*.

Communication between the interface and the Microdata computer is straightforward. The paper tape read instruction is executed when the CPU is ready for the next byte of data. It clears the accumulator, reads one byte from the reader into the accumulator and advances the paper tape one frame. With the interface in place, the only difference is that the read pointer in the buffer is incremented instead of advancing the paper tape. Eight bits of data are then transferred into the least significant eight bits of the accumulator in time for the next instruction to use it. The Microdata must then wait three milliseconds before the next read can be requested. This delay is so that the interface logic has time to settle. This is not a problem because there is usually a significant amount of processing to do before the program is ready for the next byte, although it is something which the Microdata program must guarantee. If you do not wait long enough the byte read may be the same as the previous one and the logic may get into a bad enough state to cause the Microdata to halt.

3.5. A Caveat

Anyone who has used the Photon will note that there is no longer any capability to read paper tapes. The reader was in an unrepairable state in late 1977 and because of the prohibitive cost of buying a new unit, it was simply removed. Since the interface had been in use for over a year and was able to perform all of the functions of the paper tape reader, typesetting was still possible and in fact easier due to the reduction in the number of procedures a user had to perform. The fact that the only software system being used was the system

---

* There is one case where the software for PROFF will stop if it reads an unintended tape feed, but experience shows that this does not happen in practice. The hardware bootstrap loader does not ignore nulls, but this problem is solved by using a software high core loader, to be discussed in a later section.

provided by Photon meant that booting was not necessary unless core memory was overwritten.

When the reader broke, the software for the Microdata was still being loaded by paper tape when necessary. Only typesetting codes were transferred via the interface. No one had thought to utilize the interface for program loading as it was rarely performed. It was clear that some method would have to be devised to reload the software if the image in memory was damaged. It became even more critical when the image was actually destroyed. After about six months downtime due to not being able to reload the software such a method was found. This method compensated for the interface not transmitting certain bytes but did not completely solve the problems of synchronizing the Microdata and the Honeywell due mainly to the Honeywell not responding fast enough as discussed in section 3.4. This meant that the software could not be loaded when the Honeywell load was heavy.

Various techniques were used to speed up the Honeywell's response but the problem never fully went away. The new software system described in this essay does solve the protocol problem; more will be said about how it is done in the following sections.

Persons who used the Photon before these changes mentioned in the previous sections should be aware that there have been major changes to the interface. In summary, the changes are as follows:

1) An empty buffer yields nulls as described above. This is only a partial solution to the empty buffer problem, alternatives are discussed in section 4.8.

2) The EOM byte now does not go into the buffer and the Microdata is able to read from the buffer at any time. The EOM byte used to go into the buffer and as soon as it was seen by the interface logic it was impossible for the Microdata to read bytes from the buffer. This caused some files to be terminated improperly, and the beginnings of subsequent files to have extra bytes at the beginning.

These changes were made because we believed that there were errors in the original design; the original specifications did not anticipate all the problems that were subsequently encountered. Solutions are proposed in section 4.8 based on 20/20 hindsight for the problems mentioned here and in the previous sections.

4. The Software System

4.1. Some Terminology

Now that the various hardware components have been described, the software needed to produce a typeset document from a source file will be considered. A note of clarification concerning the various software packages is in order. The programs which process files of text interspersed with formatting commands are called *formatters* and will be referred to by the name of the software package (PROFF, TROFF or TYPE). The exception to this is that we will usually use TROFF to mean the use of any combination of TBL, EQN and TROFF. On the other hand, the programs that run on the Microdata to process PROFF, TYPE or TROFF output and drive the photo unit will be called *Econosetter programs*. The Econosetter program for PROFF will also be referred to as the *737 program* since it was the original code supplied by Photon.

The three formatters differ significantly in their method of operation and capabilities. In the next section we explore these differences and the effects on the corresponding Econosetter programs.

4.2. The Formatters (PROFF, TROFF and TYPE)

4.2.1. PROFF

The Photon machine was designed to be used as a classical typesetter as discussed in section 1.1. When it was acquired, the 737 program was provided on paper tape suitable for loading into the Microdata via the hardware bootstrap loader. It was intended that the user would be able to punch his document directly onto paper tape using commands that the 737 program accepted as input. No other formatting program was needed, as the 737 program handled all processing. Because of this functionality, typesetting proceeds slower than if the formatting had been done elsewhere. Also, because of the limited core memory of the Microdata, the formatting that can be done does not include some of the more sophisticated techniques such as footnotes and page breaks.

PROFF was written as a technology bridge, allowing users familiar with line-printer formatters to be able to typeset their documents without learning the formatting language provided by Photon. PROFF was also designed to be compatible with ROFF, the general purpose formatter for line printers used extensively on the Honeywell. In fact, PROFF owes its name to ROFF, being derived from the name Photon ROFF. Apart from this similarity in name and the compatible input commands, the two programs perform entirerly different functions. Whereas ROFF is a formatter, PROFF is a preprocessor which translates its input to appropriate TTS codes for use with the 737 program provided by Photon.

Appropriate macros are provided in PROFF so that users can debug their commands by producing a line-printer version of the document, thus conserving valuable Photon resources. Previously ROFFed documents can be typeset with little or no change to the input file. Further information on the 737 program and on TTS codes is available in Section IV of [1]. Information on PROFF can be found in [2], which is a tutorial and reference manual.

4.2.2. TROFF

The TROFF program is a locally modified version of the Bell Labs formatting program. The design of the output codes is simple, yet it provides great power and flexibility. A summary of these codes is given in Appendix C. TROFF is compatible with the line-printer formatting program NROFF (a variant of ROFF) but the functionality of TROFF is very different from that of PROFF as TROFF does its own formatting.

There was no reasonable way that the output from TROFF could be typeset on the original Photon, as the 737 program was not designed to allow control of the typesetter in any straight-forward manner. Work was started on such a low-level program, but this task proved harder than first imagined. The documentation in the manuals provided by Photon for the I/O instructions is almost non-existent and the Photon Company had to be contacted directly to clarify most of the codes. After long and tedious work, mostly involving trial and error, methods to control the various components of the typesetter were discovered. This and some intelligent extrapolation from the 737 program provided enough information to write an Econosetter program for TROFF, which in a modified form is the program currently being used. Unfortunately this work was never documented so the steps leading up to the Econosetter program for TROFF are unknown to the author.

This program for TROFF was developed on UNIX with the aid of PHO, a locally written assembler. A description of the PHO language is given in Appendix E.

There is one other program used in conjunction with TROFF, the previewer. It allows a user to view a document on a graphics terminal before typesetting it. The advantage of this over debugging on a line printer is its interactive nature and the fact that the document can be shown in facsimile (for example, Greek letters and mathematical formulae).

The previewer displays TROFF output simulating the actions of the typesetter. A user can specify a start and end position in his document so that only certain sections are previewed. The program can also act as a simple editor when used in conjunction with the selection commands and an option to leave its output in the form of a TROFF output file. Thus whole documents can be stored as one piece and if just one page is needed it can be typeset separately. The program is available on Honeywell under 'troff/preview' and the explain file is under 'expl/trof/prev'. (See Appendix A.)

The Econosetter program for TROFF is listed in Appendix D. It can also be found on UNIX in '/usr/src/troff/photon/ntroff.p'. This is a modified version of the original program with the capability to reset the typesetter, include colour information (for the previewer) and page separators (also for the previewer).

4.2.3. TYPE

TYPE is a locally writtten package using macro-style formatting commands. Like TROFF it is a powerful formatter that can be used with many typesetters. The Econosetter program for TYPE was originally a slightly modified version of the 737 program and the program TYPE generated TTS codes. This modified

Econosetter program became unmaintainable during 1979. Because of this, TYPE was converted to generate output codes compatible with the TROFF Econosetter program. Documentation for this formatter can be found on Honeywell under 'typeset/expl/type/manual'.

4.2.4. Selecting a Formatter

The trade-offs between the different types of formatters are fairly simple. TROFF and TYPE, being very general and very powerful formatters are large and consume considerable resources formatting a text file. Once they have finished, typesetting can proceed rapidly. PROFF is very fast because it is only a preprocessor. Most of the processing must be done in the Microdata minicomputer, so restrictions must be placed on the power of the formatting language. TROFF is probably the best documented and is actually the only 'supported' software package. It is also the only system available on both the UNIX and Honeywell systems.

4.3. UNIX Programs and Procedures

The only procedure requiring UNIX is assembling the Econosetter program for TROFF. The assembled code is then sent to the Honeywell system for use on the Photon. The program PHO assembles the source file and creates a UNIX-style object module. This makes the output of PHO compatible with other UNIX software, but incompatible with the Photon. A detailed description of PHO appears in Appendix E.

The output of PHO must be translated before it can be sent to the Honeywell system. This is accomplished by a C program on UNIX. The input section of this program strips control and loader information from the UNIX object module and produces a stream of bytes which is the core image of the assembled code. The order of the bytes is: least significant byte of word 0, most significant byte of word 0, least significant byte of word 1, most significant byte of word 1, and so on. The output section then maps these bytes so that they can be sent via the Daemon* to the Honeywell. Care must be taken in sending files this way as only text files are guaranteed to be transmitted correctly bewteen the two syetems.

Since any possible 8-bit value may occur in the object form of a program, the data is not sent directly, but instead it is first translated to a text file (containing only the printable ASCII characters) which can be transferred reliably. Each byte is split into two four-bit pieces and offset by the integer value of the character 'a'. Since a four-bit field has a value of at most 15 the resulting bytes have values in the range 'a' to 'p'. The character ASCII NL (newline) is added after every 100 output bytes (50 actual bytes of program data, 25 sixteen bit words) to keep the line length compatible with the buffers in TSS. This stream of bytes is then written to the UNIX standard output which is sent to the Daemon program by the 'hsend' command, resulting in the translated assembler program being transferred to the Honeywell.

---

* See section 1.2.

The object module for this program is under '/usr/src/troff/photon/send'. It reads the standard input and writes to the standard output device. The source is found on UNIX under '/usr/src/troff/photon/send.c' and on Honeywell under 'photon/src/c/send.c'.

4.4. Some Problems on the Honeywell

The inverse mapping must be performed on the Honeywell to recover the original core image of the translated assembler program. Pairs of bytes must be reversed because the Photon loader expects the most signifiacant eight bits to come first. Also we must assure that the file does not contain any of the bytes that the interface considers to be special switch commands. Two separate problems arise in this process.

The first problem is that the interface will not transmit to the typesetter the EOL, EOM or SOM bytes correctly if they are included as data. The second problem is that if the transmission of data ever gets behind (that is, the buffer is empty when the loader wants another byte) an ASCII NUL will be read and stored (incorrectly) into memory as part of the program. The first problem is handled by patching the assembled code where the unsendable bytes occur, the second is solved by choosing another 'special' byte that when read will indicate that the buffer is empty. We then arrange to fix up the data so that none of these special bytes are ever transmitted. A more detailed description of this method is given in the next section.

4.5. Preparing a Program for the High-Core Loader

A program was written for the Microdata which simulates the hardware bootstrap loader exactly, except that it ignores the ASCII NUL byte, reducing the problem to loading this software loader and then transmitting the actual Econosetter program without using the bytes NUL, EOL, EOM or SOM. This may sound like pushing the empty buffer problem one level deeper as we first must load this new program into core, but in fact it is not. Because the loader can be written in less than 128 words (256 bytes) it can be transferred to the interface buffer in one block. Thus there is never a chance that the buffer will be empty during transmission of this program.

After the software loader is in the Microdata we must move the loader into high core. This is because the Econosetter programs for TROFF and PROFF both start at core location 0000, exactly where the software loader has just been stored. Fortunately, neither uses the last 128 words of core. We thus make the loader resident in high core, (from 0FAF to 0FFF). This is achieved by having the software loader transfer itself to high core. Finally, we need another special byte to terminate the loading process. The byte octal 240 which we will call EOB (End Of Boot) was chosen after analysis of the minimum frequency of occurrences of the bytes in the Econosetter programs for PROFF and TROFF.

Programs can be loaded correctly and reliably if the data being transmitted does not contain any of the bytes NUL, EOL, EOM, SOM or EOB. Clearly these bytes might appear in Econosetter programs, particularly NUL which is quite common. We do not want to burden future designers with the chore of removing all occurrences of these bytes.

What was done in the past is to 'patch' the code. This involves setting the low twelve bits of the word containing the illegal byte to some pattern that can be sent to the interface buffer. At the end of the actual assembler code, special instructions are generated to reset these twelve bits to their intended value. This is the method we have chosen to prepare files for transferring via the interface. In general we use a three instruction patch; a load immediate of a quantity that can be sent, an add immediate so that the unsendable twelve bits are in the accumulator and finally a store to set the proper word to its intended value. These patches are kept together in straight line code at the end of the actual Econosetter program. The program is finally set up so that it first executes these patches and halts leaving the desired program is in core. Still, if we attempted to patch all the occurrences of the NUL byte using this method the resulting code would grow too large for the Microdata's memory.

The method used to solve this dilemma is to offset all bytes that are sent by the operating system and have the high-core loader remove the offset. This still means that a certain five bytes have to be patched but the offset can be chosen to make the number of patches reasonable. This is the case when all the bytes have an offset of +3 added before an Econosetter program is transmitted to the Microdata. The high-core loader simply subtracts this same offset before it stores the byte.

The process of translating a file that has been sent from UNIX, then offsetting and patching the code so that it can be sent correctly via the interface, is performed on the Honeywell by the B program 'photon/prepare'. This program will also read a file that has already been reconstructed from the coded UNIX file. In particular, since the Econosetter program for PROFF has no PHO source, the offset and patched version is created from the Honeywell file. The difference between the two types of files is described in detail in the explain file for this program.

The program also has the capability of executing the patches in an existing program so that patches in an older version can be removed. (Remember, after the bytes are offset the original patches are not needed.) The program allows the user to specify bytes to be patched on top of the usual EOL, EOM and SOM, and the offset to use. There is also an option to produce a disassembled listing of the program for debugging purposes. The source and explain files are on Honeywell under 'photon/src/b/prepare.b' and 'expl/phot/prep'. They are also included in this report in Appendix D (source) and Appendix A (explain file). It should be noted that this program is just for preparing an Econosetter program for use with the new software and is not needed for preparing a file to be typeset. It should never be necessary to use it unless the entire typesetting system changes or new Microdata programs are written.

4.6. The Boot Procedure

Up to this point we have been concerned with writing software so that the user's task becomes as simple and reliable as possible. Someone unconcerned with designing new software for the Microdata need never worry about the preceding discussion. The peculiarities of the UNIX-to-Honeywell file transfer, the offsetting and the patching of code are all transparent to the typesetting user; the

only procedures he has to know are how to boot the Photon and how to pass a file to it. Detailed explanations of these procedures as well as their user interface are in the associated explain files which are online on Honeywell. They are also included in Appendix A.

We will now look at the operation of these two programs as well as the design decisions made when they were developed. The source for these programs is on Honeywell under 'photon/src/b/boot.b' and 'photon/src/b/pass.b'. A listing of the source is included in Appendix D.

The boot procedure loads the appropriate Econosetter program. The first step is to load the high-core loader using the hardware loader. The software loader is then executed and either the PROFF or TROFF Econosetter code is transferred. When the transmission is complete the code EOB is sent to stop the loading process. The user then hits RESTART to cause the required patches to be performed.

As mentioned in the description of the hardware the core memory retains its contents even when powered down. Since the Econosetter program for TROFF is entirely contained in the first 2K words it is only necessary to load the part of the PROFF program that could have been overwritten. This gives us a four-fold savings in the time required to boot for PROFF. Of course it is possible to disturb the high core portion of the 737 program (either maliciously or accidentally) so an option to transfer the full program is provided in the boot command. Since most boots that fail are caused by the hardware RESET and RESTART switches not working, one should make sure this is not the case before using this option.

4.7. The Pass Procedure

The pass program takes the output from TROFF or PROFF and sends it to the interface to be read by the Microdata. It assumes that the correct Econosetter program has been booted. The only special procedure it performs is to ensure that none of the bytes EOL, EOM or SOM appear in the transmission. This can be done using the parity bit as discussed in the interface section, as both Econosetter programs ignore the high bit of a byte.

There used to be two separate programs to send TROFF and PROFF output* but they performed the same function so they were combined. The only difference between sending the two types of output is that when sending a TROFF file an extra byte is sent at the very end of the file. This is used to cause the program executing in the Microdata to stop, forcing the user to hit RESET and RESTART before typesetting another file. This ensures that the carriage always starts at the proper offset. It is a trivial test for the program to automatically decide what type of output is being sent. When the TROFF formatter eventually puts out this byte itself, the code for generating it in the pass program will be removed.

---

* They were called 'photon/ts' and 'photon/pass', respectively.

Both of these user programs (boot and pass) keep a record of their usage in 'photon/etc/nbooters' and 'photon/etc/nusers'. The id of the user, the day, time and software used are all recorded. Both programs buffer bytes to be sent to the interface in groups of 127 bytes. The number 127 was chosen because it is less than or equal to 129 (see section 3.4) and because the buffer has 256 positions. This means that on start-up, after the first 127 bytes are sent there are still greater than 128 positions left in the buffer so the interface immediately sends a carriage return and another buffer can be sent right away. With this method the buffer is almost completely full before we start, effectively providing double-buffered transmission.

4.8. Alternative Solutions

The preceding sections traced the development of the new software system which increased the reliability of the typesetting process. There were many decisions made along the way that may have seemed quite arbitrary to the reader. This section discusses the major decisions and some of the alternatives considered.

The one component that was already designed and built (before work was started on this essay) was the interface and it seemed that most of the software was written to overcome its restrictions. The fact that three bytes (SOM, EOM and EOL) cannot be used seems very restrictive. An obvious improvement is to have an escape character along with these three special bytes. If you desired to switch the line between the terminal and the Microdata the codes could be sent normally, but if you wanted to put these codes in the buffer they could be preceeded by the escape character. If it was necessary to send the escape as a legitimite byte you would send two in succession. The hardware for recognizing this logic does not seem to be too difficult to construct and it may even be a feasible modification to the existing interface.

Related to this is the problem of the Microdata recognizing an empty interface buffer. Clearly it would be desirable to be able to check a flag in the interface that would indicate whether there is data to be read or not. Making the READ instruction itself wait until data is available is even more preferable. This concept of waiting for data to become available seems to be beyond the design of the Photon hardware as it is based upon paper tape and inherently 'knows' when the next byte is available. Due to the nature of the Photon it seems we must be content with our 'solution' that sets aside a special byte (NUL in our case) that when read from the buffer indicates that the buffer is empty. Both the high-core loader and the Econosetter program for TROFF are written with this consideration in mind.

As mentioned in section 4.5 we set aside an extra byte to terminate the loading process. We could just as well have sent the length of the program to be transfered in the first few bytes and have the loader stop when it had read that many bytes. This has the disadvantage that certain lengths can not be sent, and their is extra processing required to set up such a scheme. Although this is a straightforward procedure the method of distinguishing an EOB byte is quicker. If a suitable byte could not be found or if it would cause too many patches, this alternative might be appropriate.

4.9. Summary

This new system is by no means perfect; it only reflects our current knowledge of the Photon. In the past the internal workings have been discovered only by tinkering, and the knowledge gained was only available through word of mouth. This made it difficult for any original work to be done. This essay has documented much of what is known about the Photon, but it should be noted that some areas are still not well understood. In particular it would be of great benefit to understand the I/O mechanism more completely; as a side effect this would lead to documentation of the software supplied by Photon. To accomplish this much experimenting needs to be done by a hardware-oriented person who can read and understand circuit diagrams as these seem to be our only source of documentation. This work would also allow the removal of any remaining bugs in the Econosetter programs and allow more effective use of the typesetter in general. It is hoped that this essay can act as a stepping stone towards such research.

5. Acknowledgements

Appreciation is expressed to John Corman, who helped me to understand the various hardware components. His patience has been sorely tried by various hardware failures. I would also like to thank Rick Beach, Kelly Booth and Johann George for their ideas on developing the software package, especially Johann's knowledge of the Photon's I/O and the many hours he spent figuring them out while writing and debugging the Econosetter program for TROFF. Finally Charles Forsyth wrote the assembler PHO which made modifications to the Econosetter programs less painful.

Rogues gallery

Interface: Walter Banks, Roger Sanderson, Rick Madter, John Corman

Early work on 737: Dr. Laurie Rodgers, Mark Brader, Rick Beach

Early PASS: Karl Boekelheide

PROFF: Rick Beach, Johann George

TROFF: Johann George, Damon Permezel

TYPE: Johann George, Bill Ince, Alex White, Rick Beach

PHO: Charles Forsyth

BOOT: Johann George, Keith Dorken

Users: Dr. Mike MacKiernan, Dr. Gaston Gonnet, Bill Ince, Rick Beach, Mark Brader, Ian Allen, various **mathNEWS** personnel, Dr. Mike Malcolm

Original idea: Dr. W. Morven Gentleman

6. References

[1] The Photon Econosetter Reference Manual

This is the original manual provided to us by Photon Incorporated and contains general desciptions, some documentation on the Microdata, an explanation of the software supplied and the circuit diagrams. The manual is usually in the possesion of John Corman in MC 3066.

[2] The PROFF Formatter - Richard J. Beach

A University of Waterloo RESEARCH REPORT CS-76-08 available in MC 5181.

Appendix A

Explain Files

This appendix lists all the explain files mentioned in this essay exactly as they appear online. They can be found on the Honeywell under the filename given at the top of the page on which they appear. The files are listed in the order in which they will probably be needed by a user. Special purpose and maintenance command descriptions are included and appear after the 'normal' user interface commands.

The Photon

Explain.......................................A.1.1
Index...A.1.2
Schedule......................................A.1.3
Boot..A.1.4
Pass..A.1.7
Help..A.1.8
List TTS Codes................................A.1.10
Send..A.1.11
Prepare.......................................A.1.12

The Previewer

Explain.......................................A.2.1

PHOTON - the PHOTON Econosetter typesetter
(32 lines follow)

Description:
The PHOTON Econosetter is a programmable analog phototypesetter, connected to and controllable from the MFCF Honeywell 66/60 computer. It contains movable disks of type styles, and exposes a roll of photographic paper, which is cut and developed. The PHOTON supports up to four different typefaces ("fonts"), each in several different sizes. The only fonts currently available on the typesetter are Times Roman, Times Italic, Times Bold, and a special Mathematics font. The smallest size of type available is 8 "points" (a "point" is 1/72 inch), and the largest is 18 points (1/4 inch).

There are several different varieties of text formatters which can be used to prepare input for the PHOTON (see below). The PHOTON itself must be programmed ("booted") to accept this input, and then the file must be transmitted from the Honeywell, using a special communications interface connected to the terminal in the PHOTON room (MC 3017). "explain photon index" will point you to more detail on the operation of the PHOTON; in particular, "expl phot boot" and "expl phot sched" explain how to "boot" the photon and schedule time for its use, respectively.

See Also:
expl phot index - PHOTON explain index
expl trof - TROFF text formatter

Files used:
photon/p/<software> - various PHOTON internal programs

(Copyright (c) 1980, University of Waterloo)

PHOTON - index
(13 lines follow)

```
Under catalog "expl pnoton":
boot      - boot the Econosetter from TSS
expl      - description of the Econosetter
help      - a short guide on the use of the Photon
index     - this index
l         - list TTS typesetting codes from file
pass      - pass a PROFF, TROFF or TYPE file to the Econosetter
prepare   - read, offset and patch a Microdata program
sched     - reserve or cancel time on the Econosetter
send      - send PHO output from UNIX to TSS
```

(Copyright (c) 1980, University of Waterloo)

photon/sched - Schedule time on the Photon Econosetter
(38 lines follow)

Syntax:
    photon/sched [-Reserve | -Cancel]

Options:
  -Reserve
    Display schedule and request time to be reserved.
  -Cancel
    Display schedule and request reserved time be cancelled.

Description:
  Photon/sched maintains a schedule of time for using the Photon Econosetter. The -Reserve option displays the schedule for the current week, and prompts you for times that you wish to reserve the typesetter according to the following rules:

1) Only 1.5 hours at one sitting between 08:00 and 24:00
2) At least 1.5 hours between sittings between 08:00 and 24:00

The program will prompt you for more reserved times until you enter "done", "quit" or an empty line.
  The -cancel option permits you to cancel a time reservation. The schedule will be displayed and you may enter the time reservation that you wish deleted.

Files used:
    photon/etc/schedule - list of dates, times and userids

Bugs:
    You cannot schedule a time which spans more than 1 day.

(Copyright (c) 1979, University of Waterloo)

photon/boot - Boot the PHOTON Econosetter from TSS
(113 lines follow)

Syntax:
    photon/boot [software-file | option]

Examples:
    photon/boot -Proff
    photon/boot -Troff
    photon/boot photon/p/newsoftware

Options:
  Defaults:
    -Proff
  -Troff
    Bootloads the TROFF software.
  -Proff
    Bootloads the standard Econosetter software supplied by Photon Inc. This is the default software loaded if no option is provided.
  -Type
    Bootloads software suitable for use with typeset/type (same as that for TROFF).
  -RST
    This options provides the capability to boot the entire PROFF software, including high core tables. It should not be used by those who do not know what this means. It takes about twice as long as the regular boot for PROFF and the final light pattern should be 06D3 (hex).
  filename
    You can specify a file containing different software. This option is for use only by those select few who know how to modify the internal software for the Photon. You must also be familiar with the code for the high core loader and this boot program. Any existing software will have to be prepared to be accepted by the new system. See jhbuccino for details.

Description:
  Photon/boot is used to reload or change the software operating in the Photon Econosetter typesetter. The bootloading procedure is outlined below.
  To begin, ensure the typesetter has all switches set to their "green dot" positions. You will have to open the small panel on the front cover to reach the LOAD switch and to see the indicator lights. All the black toggle switches on the Microdata panel should be in their "up" position except SENSE 3 which must be in the "down" position for the boot procedure to work. After typing the boot command you will

be asked to hit the break key on the terminal. This ensures that the interface buffer is emptied before the software is sent to it. After this you will be requested to "Press RESET and LOAD". The RESET and LOAD switches are located inside the small front cover. The row of lights should immediately begin "counting". If not, the boot sequence should be restarted by hitting the break key on the terminal and the RESET switch on the typesetter. A small letter "j" will appear to overwrite the word "Press". This is a normal part of the bootload sequence.

About 10 seconds after the LOAD switch is pressed, a message will appear on the terminal asking you to "Press RESET and RESTART". The indicator lights will still be counting at this time. When RESET is hit all of these lights should go out, and after RESTART is hit the lights should indicate 0069 (lights 1, 4, 6 and 7). You must then check that the lights do show this pattern. If they are correct you simply hit the "return" key to continue. If for any reason they are incorrect you must hit break and the boot procedure must be repeated from the start. After the "return" key is hit you will then be asked to "Press RESET and RESTART again". When RESET is hit all the indicator lights must go out, and then RESTART will cause the STOP light (on the upper panel) to go out and as cause the indicator lights to flicker. Again, the small j is printed just before the word Press.

Depending on whether you are booting TROFF or PROFF the STOP light will remain off for about 30 or 50 seconds respectively. After waiting this amount of time the STOP light should come on and the indicator lights should show 0FC9 (lights 1, 4, 7, 8, 9, 19, 11 and 12). If the lights show anything else or the STOP light does not come on after a reasonable amount of time the boot procedure must be repeated from the start. You can now follow the instruction "Press RESTART". If you are bootloading the PROFF software then hitting RESTART should cause the lights to show 06E8 (lights 4, 6, 7, 8, 10 and 11). Otherwise you are bootloading the TROFF software and hitting RESTART should cause the lights to show 0321 (lights 1, 6, 9 and 10). If after hitting RESTART the lights do not show correctly (as indicated above) then you should repeat the bootload sequence.

To check if the typesetter is working, try typesetting a small file and either listen for the "right" noises or develop the output. Sample test files are in photon/test/proff and photon/test/troff. Output created by these files is be posted in the typesetter room.

The date, time, your userid and the software loaded are recorded in a statistics file after successful bootloading.

expl/phot/boot

Files used:

photon/etc/nbooters	- statistics file
photon/p	- catalog of Econosetter programs
photon/test	- catalog of test files

Bugs:

The RESET and RESTART toggle switches on the Photon do not always work when first pressed. You can tell that the RESET worked by observing that the indicator lights all go out. If they do not you must hit RESET again. The RESTART toggle will cause the STOP light to go out and the indicator lights to flicker and/or change.

Comments and suggestions to userid jhbuccino.

(Copyright (c) 1980, University of Waterloo)

photon/pass - Send a PROFF, TROFF or TYPE file to the Econosetter
(35 lines follow)

Syntax:
     photon/pass filename

Description:
     This command transfers the output from the various formatters to the Photon Econosetter typesetter. The typesetter must be loaded with the appropriate software (see "expl photon boot").
     You will first be requested to hit the break key on the terminal to ensure that the interface buffer is cleared. The command will then prompt "Press RESET and RESTART" which requests you to press the RESET and RESTART switches on the typesetter. You should hear the mirror carriage in the typesetter move to its home position when you press RESET. The typesetter should begin making noises when you press the RESTART switch. It is a good practice to wait until you see the small j print just before the word Press (indicating the file is being sent) before you press the RESTART switch.
     You may interrupt the transmission by hitting the BREAK key on the terminal. You may have to wait a short while until the typesetter stops, and the usage information is recorded by photon/pass.

Files used:
     photon/etc/nusers - statistics file for typesetting usage

Bugs:
     It is sometimes possible for the transmission to get scrambled and the line hangs until BREAK is hit. If the system crashes or your line disconnects, you don't see the message as the typesetter trys to typeset it!

(Copyright (c) 1979, University of Waterloo)

How to use the Econosetter

Power Up Procedures

1. Turn the PWR ON switch to the ON position.
2. Wait five seconds and switch the DISC ON to the on position.
3. Power up the terminal and sign on to TSS as usual.
4. Ensure that all switches on the top panel of the Photon are in their 'green dot' position.
5. If the cannister for collecting the paper is not mounted over the output slot remove it from the developer and mount it.
6. Open the door to expose the Microdata panel. All the toggles must be in their 'up' position except that the SENSE 3 toggle must be in the 'down' position.
7. Turn the developer and stabilizer unit on.
8. Determine by looking at the card that indicates which softawre package is currently booted whether it is necessary to re-boot or not. If you don't trust the card try passing a test file.

Boot Procedure

1. Type the command 'photon/boot -t' to boot TROFF (or TYPE) and simply 'photon/boot' for PROFF.
2. Follow the instuctions given.

Pass Procedure

1. Type the command 'photon/pass filename' on the terminal where filename is the output from PROFF, TROFF or TYPE.
2. Follow the instructions given.

Developing Procedures

1. After advancing the paper cut it with the built-in paper cutter.
2. Remove the cannister (being careful that the exposed paper remains in the unit) and place it in the input section of the developer.
3. Edge the paper forward until it engages the rollers.
4. Close the casing and when the paper starts to emerge from the rollers, help guide the paper out.
5. When the document is free of the rollers lay it aside to dry.
6. Replace the cannister on top of the Photon.

Powering Down

1. Turn off the developer.
2. Turn the DISC ON switch to the OFF position.
3. Sign off of TSS and turn the terminal off.
4. Wait five seconds and then turn the PWR ON switch off.
5. Turn the lights out in the room and make sure the door is locked.

photon/l - list Photon typesetter codes (TTS codes)

Syntax:
     photon/l <input-file> <output-file>

Description:
     Photon/l is useful only for those people who understand what TTS codes are. TTS codes are generated by PROFF and are rather inscrutable. Documentation on TTS codes may be found in the Photon manual "Phase 2 Software Application Handbook" and are summarized in the PROFF manual.

(Copyright (c) 1979, University of Waterloo)

send - translate PHO output to be sent by the UNIX daemon
(30 lines follow)

Syntax:
    /usr/src/troff/photon/send

Examples:
    /usr/src/troff/photon/send <a.out | hsend

Description:
    This is a UNIX program written in C that takes a UNIX style object module generated by the Photon assembler PHO, strips it of administrative and loader information, then translates the remaining bytes so that they can be sent to the Honeywell timesharing system TSS (via the UNIX daemon). The mapping of the bytes can be described as follows. After the input file is stripped (as above) we are left with a stream of bytes which is the actual assembled code. Two consecutive bytes form a Photon word, the first byte being the low eight bits of the word. Each byte is now split into two four bit sections each having a value in the range 0 - 15 (a hex digit). Now each of these new bytes has the value 'a' added to it. This process leaves a stream of bytes twice as long as the original file. All the bytes in the output file are in the range 'a' - 'p'. Newlines are inserted after every 100 bytes written out to keep the line lengths reasonable.
    At the completion of this program the standard output can be sent via the UNIX daemon to Honeywell for use with the program 'photon/prepare'.

(Copyright (c) 1980, University of Waterloo)

photon/prepare - prepare a program for the photon/boot command
(67 lines follow)

Syntax:
```
     photon/prepare input-file [options]
```

Examples:
```
     photon/prepare pipe.end -u pc=0 pc=240 >/software
     photon/prepare /software oo=3 -p -nf -d >out
```

Options:

Defaults:

OldOffset=0 NewOffset=3

-Unix

The input file is in UNIX format. (see below)

OldOffset=n

The input file's bytes have an offset of n. Unix format files cannot have a non-zero offset.

-Patch

The input file has standard patches that should be performed before the output stage is entered.

-NoFix

The program will not attempt to offset and patch the input file when this option is given. It is useful with the '-Diss' option.

NewOffset=n

Add n to each byte of the code before deciding which bytes to patch.

-Exec

Produce code so that after the patch code is executed on the Microdata the program starts execution immediately. (by a jump to location 0000)

PatchChar=n

Do not allow the byte with value n to occur in the output file by 'patching' the code (if necessary).

-Diss

Produce a dissasembled version of the output code.

Description:

Photon/prepare should only be used by those people who appreciate the problems of developing new software on UNIX for the Microdata minicomputer in the Photon Econosetter. Basically the code for such a program must be massaged into a form that can be used with the command photon/boot. The documentation for these procedures can be obtained from Rick Beach (userid rjbeach).

This program takes two basic forms of input; UNIX format and Honeywell format. A program with UNIX format is assumed to have each word of the Microdata program represented by four consecutive bytes. These bytes each represent four

bits of a Microdata word. Each byte has a value in the range 'a' to 'p' which maps in the obvoius manner (by subtracting 'a') to a binary value between 0 and 15. If the bytes are labelled 1, 2, 3 and 4 with each containing a hex digit, the resulting Microdata data word is given by 3412.

The Honeywell format (which is the default input format) has two consectutive bytes representing a Microdata word. The resulting 16 bit word is just the first byte shifted up 8 bits and or'ed with the second byte. These bytes can also have an offset (OldOffset) which is subtracted first before they are combined into words.

After the words are reconstructed and stored in core the existing patches can be performed. The code is then fixed (if desired) using the NewOffset any the PatchChar's and finally output in Honeywell format. The option '-Diss' can be used to get a listing of the input program by specifying the '-NoFix' option. The patches created by this program normally end with a halt instruction but this can be changed with the '-Exec' option.

```
troff/preview - display TROFF files on graphics terminal
              - edit TROFF files destined for the Econosetter
(76 lines follow)

Syntax:
     troff/preview [options]* filename

     -TEKtronics              -HewlettPackard
     -Econo
     -White                   -Black
     Start=<nn>               End=<nn>
     BauDrate=<nnnn>

Examples:
     troff/preview -tek bd=9600 t.out
     troff/preview s=5.2 e=11.6 file1 -e >newfile
     troff/preview myfile

Options:
   Defaults:
       -HP BauDrate=1200 -White Start=0 End=0 [EOF]
   -TEKtronix
       Output will be to a Tektronix 4010.
   -HP
       Output will be to a HP2648A terminal.
   -Econo
       Output will have the form of  a  TROFF  output  file.
       This,  in  conjunction with the Start and End options
       allow typesetting of sections of large documents.
   -White
       Output will be black print on white  if  on  HP  ter-
       minal.
   -Black
       Output will be white characters on black background.
   BauDrate=<nnnn>
       This  allows  the  program  to put out an appropriate
       number of synchronization characters so that  nothing
       is  lost during transmission of data to the terminal.
       The default rate is 1200.
   Start=<nn>
       Allows viewing to start <nn> inches  into  the  docu-
       ment.  The value <nn> may be integer or real.
   End=<nn>
       Allows  viewing to end <nn> inches into the document.
       The value <nn> may be integer or real.

Description:
     This   program   allows   the  previewing  of  a  TROFF
typesetting output file on a Tektronix  4010  or  a  HP2648A
graphics terminal, before attempting to typeset it using the
```

Econosetter. It also allows elementary editting of the file allowing portions to be typeset. To do this you must specify the -Econo option and direct the output to a file. If a HP terminal is used, black print on a white screen is the default, overridable with the -Black option to give standard white letters on a dark screen. If a Tektronix 4010 is selected, this option has no effect.

After each terminal page is completed, a prompt in the form of a 'cntl-g' (bell) is issued and the user may hit carriage return to view the next terminal page or type a '+' or a '-' follwed by a integer to indicate skipping or reviewing terminal pages. All numbers specified, whether on the command line or after a prompt indicate document inches. (One terminal page = (approx.) 4 inches of a typeset page.)

See Also:

    expl troff index   - TROFF index
    expl photon index   - PHOTON Econosetter index

Bugs:

Due to rounding error and the resolution of the HP terminal, alternate lines may appear to change in point size when, in fact, they are identical.

The character set displayed could be much better; large point sizes look ugly.

The BauDrate code is not perfect. You may still get frogged output at 9600 baud.

(Copyright (c) 1980, University of Waterloo)

Appendix B

Regenerating the Software System

1. You will need the following files.

```
UNIX:  /usr/src/troff/photon/send.c
       /usr/src/troff/photon/ntroff.p
       /usr/src/troff/photon/hcldr.p
TSS:   photon/src/b/boot.b
       photon/src/b/pass.b
       photon/src/b/prepare.b
       photon/p/proff
       photon/p/rstpt
```

These files can be restored as of January 10, 1980 if they are not online or if you suspect they have been damaged.

2. On UNIX, type the following commands.

```
cc /usr/src/troff/photon/send.c
mv a.out /user/src/troff/photon/send
pho /usr/src/troff/photon/ntroff.p
/usr/src/troff/photon/send <a.out >troff
pho /usr/src/troff/photon/hcldr.p
/usr/src/troff/photon/send <a.out >hcldr
hsend troff hcldr
```

This command sequence is also in the file /usr/src/troff/photon/reg and can be executed using the command 'sh'.

3. When the file transfer from UNIX is complete, execute the following TSS commands.

```
b photon/src/b/boot.b -d h=photon/boot
b photon/src/b/pass.b -d h=photon/pass
b photon/src/b/prepare.b -d h=photon/prepare
photon/prepare hcldr -u no=0 >photon/p/hcldr
photon/prepare troff -u pc=0 pc=240 >photon/p/ntroff
```

This command sequence is also in the file photon/make/reg and can be executed using the 'ec' command.

4. The system should now function as documented.

Appendix C

TROFF Output

This appendix explains the commands that TROFF outputs to drive the Photon. There are two basic types, the commands that control pointsize, font selection, the carriage and paper motion, and the commands that select the character to be displayed. TROFF always outputs seven-bit codes (octal 000 - 177). The first type of commands uses the codes 000 - 017, the codes 020 - 177 are reserved for character selection.

A code in the range 000 - 020 is always followed by some number of eight-bit bytes that act as the argument to the command. For instance, the change font command (001) is followed by one byte that must have a value in the range 0 - 3 which selects one of the four available fonts. A complete table of these commands, their arguments and functions is given below. The arguments are represented by the notation *[i]* in the table and each represents eight bits. On the next page is a table of all the available characters from TROFF. The blank spaces in the table have corresponding characters on the character disc, but they are not available from TROFF at this time.

Command		Function
000		NOP – The Econosetter program for TROFF ignores NUL bytes.
001	*[i]*	Set Font – The argument must be in the range 0 - 3 selecting one of the four fonts (0 = Roman, 1 = Italic, 2 = Bold, 3 = Math).
002	*[i]*	Set Direction of Carriage Motor – The argument must be in the range -1 to +1 (-1 = left motion, 0 = no motion, +1 = right motion).
003	*[i]*	Set Pointsize – The argument must be in the range 0 - 3 selecting one of the four pointsizes (0 = 8 points, 1 = 10 points, 2 = 14 points, 3 = 18 points).
004	*[i]*	Horizontal Motion – The carriage is moved $i/432$ inches in the current direction where i is the argument in the range -64 to +63.
005	*[i]*	Vertical Motion – The photographic material is moved $i/144$ inches where i is the argument in the range -64 to +63.
006		Reset – A hardware RESET is simulated.
007	*[i]*	Set Colour – The argument selects the colour to draw the output in. This is a NOP on the Photon, but the Tek 4027 previewer uses it.
010	*[i] [j]*	Page Mark – The argument forms a sixteen-bit integer that indicates the page number relative to the beginning of the output. This is a NOP on the Photon, but the previewer can use it.
011		Halt – Causes the Photon to execute a HALT instruction.

Octal Code	Font: Roman	Italic	Bold	Math
020	1	1	1	→
021	2	2	2	←
022	3	3	3	↓
023	4	4	4	↑
024	5	5	5	√
025	6	6	6	‾
026	7	7	7	⩾
027	8	8	8	⩽
030	9	9	9	≡
031	0	0	0	≈
032	$	$	$	
033	)	)	)	~
034	-	-	-	≃
035	e	e	e	γ
036	t	t	t	ϵ
037	a	a	a	κ
040	i	i	i	η
041	n	n	n	ω
042	s	s	s	λ
043	o	o	o	θ
044	r	r	r	δ
045	h	h	h	π
046	d	d	d	μ
047	l	l	l	ς
050	u	u	u	ξ
051	c	c	c	φ
052	f	f	f	ν
053	m	m	m	
054	y	y	y	ζ
055	w	w	w	β
056	p	p	p	ι
057	g	g	g	ο
060	b	b	b	ψ
061	v	v	v	χ
062	k	k	k	σ
063	q	q	q	α
064	j	j	j	ρ
065	x	x	x	υ
066	z	z	z	τ
067	,	,	,	∪
070	.	.	.	⊂
071	;	;	;	
072	'	'	'	
073	#	#	#	⊆
074	]	]	]	}
075				
076	/	/	/	
077	—	—	—	|
100	¼	¼	¼	⎧
101	½	½	½	⎪
102	¾	¾	¾	⎨
103	%	%	%	⎩
104				⎫
105				⎬
106				⎭
107				⎡

Octal Code	Font: Roman	Italic	Bold	Math
110	&	&	&	⎣
111	?	?	?	⎤
112	!	!	!	
113	(	(	(	≠
114	+	+	+	⎦
115	E	E	E	Θ
116	T	T	T	Ξ
117	A	A	A	Ω
120	I	I	I	ϒ
121	N	N	N	
122	S	S	S	∇
123	O	O	O	Φ
124	R	R	R	Λ
125	H	H	H	∈
126	D	D	D	¬
127	L	L	L	ℰ
130	U	U	U	Σ
131	C	C	C	
132	F	F	F	∫
133	M	M	M	
134	Y	Y	Y	Π
135	W	W	W	Δ
136	P	P	P	Ψ
137	G	G	G	∅
140	B	B	B	~
141	V	V	V	
142	K	K	K	◯
143	Q	Q	Q	Γ
144	J	J	J	
145	X	X	X	>
146	Z	Z	Z	<
147				∩
150	'	'	'	⊃
151	:	:	:	
152	‘	‘	‘	
153	*	*	*	⊇
154	[	[	[	{
155	"	"	"	
156	\	\	\	
157	|	|	|	
160	fi	fi	fi	£
161	fl	fl	fl	
162	ff	ff	ff	
163	°	°	°	∘
164				÷
165				×
166	–	–	–	
167	¶	¶	¶	´
170	§	§	§	ˆ
171	○	□	□	`
172	¢	¢	¢	∝
173	†	‡	†	˜
174	=		=	±
175	®	®	®	
176	©	©	©	∞
177	@	@	@	∂

Appendix D

Source Code

The following pages contain listings of the source code for the typesetting system developed in this essay. The listings are divided into sections by programming language. The code can also be found on the Honeywell system in the filename given at the top of the page that the listing appears on.

B Source
- Boot D.1.1
- Pass D.1.7
- Prepare D.1.12

C Source
- Send D.2.1

PHO Source
- High-Core Loader D.3.1
- Troff D.3.6

```
%b/manif/.bset
%b/manif/t.drls

NUL         = 0000;     /* Indicates an empty interface buffer. */
EOL         = 0021;     /* End of line character sent by datanet. */
EOM         = 0331;     /* Turns interface back to the terminal. */
SOM         = 0352;     /* Turns interface to the Photon. */
EOB         = 0360;     /* Indicates end of boot. */

BUFFER      = 255;      /* Size of interface buffer (in bytes). */
SEND        = 127;      /* Number of bytes to send at a time. */
STD         = -4;       /* Error messages go to the terminal. */
TRM         = -5;       /* Force read from the terminal. */

TRPATCH     = 01441;    /* Troff stops at 0321 (hex) after patches. */
PRPATCH     = 03350;    /* Proff stops at 06E8 (hex) after patches. */
RSPATCH     = 03323;    /* Reset stops at 06D3 (hex) after patches. */

HCLDR       = "photon/p/hcldr";
TRSOFT      = "photon/p/ntroff";
PRSOFT      = "photon/p/nproff";
RSSOFT      = "photon/p/rstpt";
STATS       = "photon/etc/nbooters$nnnn";

FILENAME    = 0;
COMMAND     = 1;
TROFF       = 2;
PROFF       = 3;
RSTPT       = 4;

file;               /* Where the core image to be booted is stored. */
optable[]
    "BOOT"      , COMM_KWD,
    "Troff"     , DASH_KWD,
    "Proff"     , DASH_KWD,
    "RST"           , DASH_KWD,
    -1;
```

```
main( argc, argv )
  {
    extrn prthex, drl.q;
    auto c, lights, buffer, count;

    lights = setup( argc, argv );
    buffer = getvec( ( BUFFER + 3 ) / 4 - 1 );

    drl.q = 0102000000;
    drl.drl( T.SETS_ );

    nobrks( 2 );
    printf( STD, "Hit *"break*" to clear the interface.*n" );
    flush();
    while( !nobrks() );
    nobrks( 0 );

    trldr( buffer );

    printf( STD, "*nPress RESET and RESTART.  The lights should" );
    printf( STD, " now indicate  0069 (hex).*n" );
    printf( STD, "If they are correct, hit *"return*" to continue.*n" );
    c = getc( TRM );
    printf( STD, "*n Now press RESET and RESTART again.*n" );
    flush();

    count = trdrv( buffer );

    printf( STD, "*nThe lights should now read 0FC9 (hex).*n" );
    printf( STD, "Press RESTART.  (Do not press RESET.)*n" );
    if( lights )
      {
        printf( STD, "The lights should now read %y ", prthex, lights );
        printf( STD, "(hex) which indicates a successful boot.*n" );
      }

    drl.q = 0102000000;
    drl.drl( T.RSTS_ );

    stats( count );
  }


setup( argc, argv )
  {
    extrn file;
    auto i, info, out, lights, driver;

    file = PRSOFT;
    out = 0;
```

```
    lights = PRPATCH;
    driver = "Proff";

    for( i = 1; ( info = argv[i] ) != -1; ++i )
        switch( info >> 18 )
          {
            case COMMAND:
                break;
            case FILENAME:
                if( out )
                    error( "Only one file can be booted at a time.*n" );
                out = argv[i];
                lights = 0;
                driver = "user's";
                break;
            case TROFF:
                if( out )
                    error( "Only one file can be booted at a time.*n" );
                out = TRSOFT;
                lights = TRPATCH;
                driver = "Troff";
                break;
            case PROFF:
                if( out )
                    error( "Only one file can be booted at a time.*n" );
                out = PRSOFT;
                lights = PRPATCH;
                driver = "Proff";
                break;
            case RSTPT:
                if( out )
                    error( "Only one file can be booted at a time.*n" );
                out = RSSOFT;
                lights = RSPATCH;
                driver = "full Proff";
                break;
            default:
                printf( STD, "%s - unknown option*n", info );
                exit();
          }

    if( out )
        file = out;
    printf( STD, "Booting with %s driver.*n*n", driver );

    return( lights );
  }
```

```
trldr( buffer )
  {
    auto i, c, n;

    open( HCLDR, "r" );
    n = 0;
    while( ( c = getchar() ) != '*0' )
      {
        if( c == '*n' )
            next;
        lchar( buffer, n++, c );
      }
    close();
    printf( STD, " Press RESET and LOAD.*n" );
    flush();
    putc( SOM );
    putline( buffer, n );
    n = 0;
    for( i = 1; i <= SEND; ++i )
        lchar( buffer, i, '*0' );
    putline( buffer, SEND );
    putc( EOM );
  }

error( arg )
  {
    printf( STD, "%r", &arg );
    exit();
  }

putline( line, n )
  {
    auto tally, tallyb;

    tally = ( &tallyb << 18 ) | 1 << 6;
    tallyb = ( line << 18 ) | ( n << 6 ) | 040;
    drl.drl( T.OTIN_, &tally << 18 );
  }

putc( c )
  {
    auto tally, tallyb;

    tally = ( &tallyb << 18 ) | 1 << 6;
    tallyb = ( &c << 18 ) | 0143;
    drl.drl( T.KOUT_, ( &tally << 18 ) );
  }
```

```
trdrv( buffer )
  {
    extrn file;
    auto c, n, count;

    open( file, "r" );
    n = count = 0;
    putc( SOM );
    while( ( c = getchar() ) != '*0' )
      {
        if( c == '*n' )
            next;
        count++;
        c &= 0377;
        if( n == SEND )
          {
            putline( buffer, n );
            n = 0;
          }
        lchar( buffer, n++, c );
      }
    putline( buffer, n );
    putc( EOB );
    putc( EOM );
    close();

    return( count );
  }

prthex( num )
  {
    auto con[3], i;

    con[0] = ( num & 0170000 ) >> 12;
    con[1] = ( num &   07400 ) >>  8;
    con[2] = ( num &    0360 ) >>  4;
    con[3] = ( num &     017 ) >>  0;

    for( i = 0; i <= 3; ++i )
        if( con[i] > 9 )
            putchar( 'A' + con[i] - 10 );
        else
            putchar( '0' + con[i] );

  }
```

```
stats( count )
  {
    extrn file, .uid;
    auto d[3], t[3], userid[3];

    bcdasc( userid, &.uid, 12 );
    open( STATS, "a" );
    printf( "%12s %s %s %s - %10d*n",
        userid, date( d ), time( t ), file, count );
    close();
  }
```

```
%b/manif/.bset
%b/manif/t.drls

NUL         = 0000;     /* Indicates an empty interface buffer. */
EOL         = 0021;     /* End of line character sent by datanet. */
EOM         = 0331;     /* Turns interface back to the terminal. */
SOM         = 0352;     /* Turns interface to the Photon. */

BUFFER      = 255;      /* Size of interface buffer (in bytes). */
SEND        = 127;      /* Number of bytes to send at a time. */
STD         = -4;       /* Error messages go to the terminal. */

PSTART      = 0427;     /* All Proff files start with this byte. */
STOP        = 011;      /* Causes troff driver to halt. */

STATS       = "photon/etc/nusers$nnnn";

FALSE       = 0;
TRUE        = 1;

FILENAME    = 0;
COMMAND     = 1;

file;           /* Where the output file is stored. */
troff;          /* Indicates what kind of file is being typeset. */
optable[]
    "PASS"      , COMM_KWD,
    -1;
```

photon/src/b/pass.b

```
main( argc, argv )
  {
    extrn troff, drl.q;
    auto i, buffer, count, start;

    setup( argc, argv );
    buffer = getvec( ( BUFFER + 3 ) / 4 - 1 );

    drl.q = 0102000000;
    drl.drl( T.SETS_ );

    nobrks( 2 );
    printf( STD, "Hit *"break*" to clear the interface.*n" );
    flush();
    while( !nobrks() );
    nobrks( 0 );

    start = time();
    printf( STD, "*n Press RESET and RESTART.*n" );
    flush();

    count = trfile( buffer );

    if( troff )
        putc( STOP );

    for( i = 1; i <= SEND; ++i )
        lchar( buffer, i, '*0' );
    putline( buffer, SEND );

    putc( EOM );


    printf( STD, "*nYour file has been typeset.  When the STOP " );
    printf( STD, "light comes*non press RESET and hold CONT LEAD" );
    printf( STD, " for about 10 seconds.*n" );

    drl.q = 0102000000;
    drl.drl( T.RSTS_ );

    stats( start, count );
  }
```

```
setup( argc, argv )
  {
    extrn file;
    auto i, info;

    file = 0;

    for( i = 1; ( info = argv[i] ) != -1; ++i )
        switch( info >> 18 )
          {
            case COMMAND:
                break;
            case FILENAME:
                if( file )
                    error( "Only one file can be typeset at a time.*n" )
                file = argv[i];
                break;
            default:
                printf( STD, "%s - unknown option*n", info );
                exit();
          }

    if( file == 0 )
        error( "Usage: Pass <filename> [-Troff] [-Proff]*n" );

  }
```

```
trfile( buffer )
  {
    extrn file, troff;
    auto c, n, count;

    open( file, "r" );
    troff = TRUE;
    if( ( c = getchar() ) == PSTART )
        troff = FALSE;
    lchar( buffer, 0, c );
    n = count = 1;

    putc( SOM );
    while( ( c = getchar() ) != '*0' )
      {
        if( c == '*n' )
            next;
        count++;
        c &= 0377;
        if( c == NUL || c == EOL || c == EOM || c == SOM )
            c ^= 0200;
        if( n == SEND )
          {
            putline( buffer, n );
            n = 0;
          }
        lchar( buffer, n++, c );
      }
    putline( buffer, n );
    close();

    return( count );
  }

error( arg )
  {
    printf( STD, "%r", &arg );
    exit();
  }

putline( line, n )
  {
    auto tally, tallyb;

    tally = ( &tallyb << 18 ) | 1 << 6;
    tallyb = ( line << 18 ) | ( n << 6 ) | 040;
    drl.drl( T.OTIN_, &tally << 18 );
  }
```

```
putc( c )
  {
    auto tally, tallyb;

    tally = ( &tallyb << 18 ) | 1 << 6;
    tallyb = ( &c << 18 ) | 0143;
    drl.drl( T.KOUT_, ( &tally << 18 ) );
  }

stats( start, count )
  {
    extrn .uid;
    auto d[3], t[3], userid[3], end;

    end = time();
    bcdasc( userid, &.uid, 12 );
    open( STATS, "a" );
    printf( "%12s %s %s - %10d*n",
        userid, date( d ), time( t, end-start ), count );
    close();
  }
```

```
FALSE           = 0;
TRUE            = 1;

CURPATCH        = 3;          /* Number of bytes special to interface. */
MAXPATCH        = 10;         /* Maximum bytes special to all programs. */
EOL             = 0021;       /* Datanet sends this after every buffer. */
SOM             = 0352;       /* Allows interface to start receiving. */
EOM             = 0331;       /* Tells interface transmission has ended. */
OFFSET          = 3;          /* Output bytes offset to reduce fixes. */

EOF             = -1;         /* Value returned to main at end of input. */
STD             = -4;         /* Assures error messages go to terminal. */

OPCODE          = 0170000;    /* Selects the opcode of an instruction. */
OPERND          = 07777;      /* Selects the operand of an instruction. */
ADDRESS         = 07777;      /* Selects the address of an instruction. */
LOBYTE          = 0377;       /* Selects bottom 8 bits of an instruction. */
HIBYTE          = 0177400;    /* Selects top 8 bits of an instruction. */
WORD            = 0177777;    /* Converts HW words to 16 bit words. */

LI              = 0030000;    /* Load Immediate */
NOP             = 0050000;    /* No OPeration (Add 0 to AC) */
ADDI            = 0050000;    /* ADD Immediate */
ST              = 0070000;    /* STore AC */
HALT            = 0114000;    /* HALT */
BR              = 0120000;    /* BRanch */

CORESIZE        = 4096;       /* Physical size of Microdata core. */
LDRSIZE         = 128;        /* Core used by the high-core loader. */
USEABLE         = CORESIZE - LDRSIZE;

%b/manif/.bset
FILENAME        = 0;
COMMAND         = 1;
UNIX            = 2;
OLDOFFSET       = 3;
PATCH           = 4;
NOFIX           = 5;
NEWOFFSET       = 6;
EXEC            = 7;
PATCHCHAR       = 8;
DISS            = 9;
```

```
/* Externals. */

file;        /* Unit number of input file. */
unix;        /* Input was sent from UNIX. */
oldof;       /* Input was written with this as offset. */

patch;       /* Perform any patches in the input before fixing. */

nofix;       /* Do not attempt to fix the code. */
newof;       /* Use this as offset when outputting code. */
exec;        /* Jump back to location zero after fixes. */

badch;       /* Pointer to list of bytes that can not be used. */

diss;        /* Code is to be dissambled. */


core;        /* Pointer to contents of core. */
last;        /* Last core address of current program. */


optable[]
    "PREPARE"   , COMM_KWD,
    "Unix"      , DASH_KWD,
    "OldOffset" , NVAL_KWD,
    "Patch"     , DASH_KWD,
    "NoFix"     , DASH_KWD,
    "NewOffset" , NVAL_KWD,
    "Exec"      , DASH_KWD,
    "PatchChar" , NVAL_KWD,
    "Diss"      , DASH_KWD,
    -1;
```

photon/src/b/prepare.b

```
main( argc, argv )
  {
    extrn patch, nofix, core, last;
    auto i, word;

    setup( argc, argv );
    core = getvec( CORESIZE - 1 );
    for( i = 0; ( word = getword() ) != EOF; ++i )
        if( i == CORESIZE )
            error( "Program too big for core.*n" );
        else
            core[i] = word;

    if( patch )
        dopatch();
    else
        last = i - 1;

    if( !nofix )
        fix();
    output();
  }
```

```
setup( argc, argv )
  {
    extrn file, unix, oldof, patch, nofix, newof, exec, badch, diss;
    auto i, info, out;

    badch = zero( getvec( MAXPATCH ) );
    badch[0] = CURPATCH;
    badch[1] = SOM;
    badch[2] = EOM;
    badch[3] = EOL;

    file = out = oldof = 0;
    unix = patch = nofix = exec = diss = FALSE;
    newof = OFFSET;

    for( i = 1; ( info = argv[i] ) != -1; ++i )
        switch( info >> 18 )
          {
            case COMMAND:
                break;
            case FILENAME:
                if( out )
                    error( "Only one file at a time please.*n" );
                out = argv[i];
                break;
            case UNIX:
                unix = TRUE;
                break;
            case OLDOFFSET:
                oldof = *info;
                break;
            case PATCH:
                patch = TRUE;
                break;
            case NOFIX:
                nofix = TRUE;
                break;
            case NEWOFFSET:
                newof = *info;
                break;
            case EXEC:
                exec = TRUE;
                break;
            case PATCHCHAR:
                badch[++badch[0]] = *info;
                break;
            case DISS:
                diss = TRUE;
                break;
            default:
```

```
                printf( "%s - unknown option*n", info );
                exit();
            }

     if( out )
         file = open( out, "r" );
  }


/* Input routines. */

getword()
  {
    extrn unix, oldof;
    auto c, c1, c2;

    if( unix )
      {
        if( ( c = getc1() ) == '*0' )
            return( EOF );
        c2 = ( ( ( c - 'a' ) << 4 ) | ( getc1() - 'a' ) ) & LOBYTE;
        c = getc1();
        c1 = ( ( ( c - 'a' ) << 4 ) | ( getc1() - 'a' ) ) & LOBYTE;
      }
    else
      {
        if( ( c = getchar() ) == '*n' )
            return( EOF );
        c1 = ( c - oldof ) & LOBYTE;
        c2 = ( getchar() - oldof ) & LOBYTE;
      }

    c = ( c1 << 8 ) + c2;
    return( c & WORD );
  }

getc1()
  {
    auto c;

    while( ( c = getchar() ) == '*n' );
    return( c );
  }
```

```
dopatch()
  {
    extrn core, last, prthex;
    auto word, nxt, ac, upd;

    if( core[0] & OPCODE != BR )
        error( "Instruction in location 0 is not a branch.*n" );
    nxt = core[0] & ADDRESS;
    last = nxt - 1;
    while( ( word = core[nxt++] ) != HALT )
      {
        if( word == BR )
            break;

        if( word & OPCODE != LI )
          {
            printf( STD, "Error performing patches in original code." );
            printf( STD, "*nExpecting a Load Immediate...*n" );
            printf( STD, "at address %y   ", prthex, nxt-1 );
            printf( STD, "read a %y*n", prthex, word );
            exit();
          }
        ac = word & OPERND;

        word = core[nxt++];
        if( word & OPCODE != ADDI )
          {
            printf( STD, "Error performing patches in original code." );
            printf( STD, "*nExpecting an ADD Immediate...*n" );
            printf( STD, "at address %y   ", prthex, nxt-1 );
            printf( STD, "read a %y*n", prthex, word );
            exit();
          }
        ac = ac + ( word & OPERND );

        while( core[nxt] == NOP )
            nxt++;
        if( ( word = core[nxt++] ) & OPCODE != ST )
          {
            printf( STD, "Error performing patches in original code." );
            printf( STD, "*nExpecting a STore...*n" );
            printf( STD, "at address %y   ", prthex, nxt-1 );
            printf( STD, "read a %y*n", prthex, word );
            exit();
          }
        upd = word & ADDRESS;
        core[upd] = ( core[upd] & OPCODE ) + ( ac & OPERND );
      }
  }
```

```
/* Output routines. */

output()
  {
    extrn newof, diss, core, last, prthex;
    auto i;

    if( diss )
        for( i = 0; i <= last; ++i )
          {
            printf( "                              " );
            printf( "%y:    %y*n", prthex, i, prthex, core[i] );
            if( ( i + 1 ) % 16 == 0 )
                putchar( '*n' );
          }
    else
        for( i = 0; i <= last; ++i )
          {
            putchar( ( ( ( core[i] >> 8 ) + newof ) & LOBYTE ) | 0400 );
            putchar( ( (   core[i]          + newof ) & LOBYTE ) | 0400 );
          }
  }

prthex( num )
  {
    auto con[3], i;

    con[0] = ( num & 0170000 ) >> 12;
    con[1] = ( num &   07400 ) >>  8;
    con[2] = ( num &    0360 ) >>  4;
    con[3] = ( num &     017 ) >>  0;

    for( i = 0; i <= 3; ++i )
        if( con[i] > 9 )
            putchar( 'A' + con[i] - 10 );
        else
            putchar( '0' + con[i] );

  }

error( arg )
  {
    printf( STD, "%r", &arg );
    exit();
  }
```

photon/src/b/prepare.b

```
/* Routines for fixing code. */

fix()
  {
    extrn exec, core, last, prthex;
    auto i, n, core0, ncore0;

    core0 = core[0];
    if( ( core0 & OPCODE ) != BR )
        error( "Cannot fix code.  First instruction not a branch.*n" );

    n = last;
    ncore0 = BR | ( ( last + 1 ) & ADDRESS );
    while( badword( ncore0 ) )
      {
        if( badword( NOP ) )
            error( "Cannot code a NOP (5000).*n" );
        code( NOP );
      }
    ncore0 = BR | ( ( last + 1 ) & ADDRESS );

    for( i = 0; i <= n; ++i )
        if( badword( core[i] ) )
            stuff( core[i], i );

    stuff( core0, 0 );
    if( exec )
      {
        if( badword( BR ) )
            error( "Cannot code a BRanch 000  (A000).*n" );
        code( BR );
      }
    else
      {
        if( badword( HALT ) )
            error( "Cannot code a HALT (9800).*n" );
        code( HALT );
      }
    core[0] = ncore0;
  }

code( data )
  {
    extrn core, last;

    if( last == USEABLE )
        error( "Fixes cause program to overflow core.*n" );
    core[++last] = data;
  }
```

```
stuff( data, addr )
  {
    extrn core, last, prthex;
    auto i, count, ix, i1, i2, i3, j1, j2, j3;

    count = 0;
    ix = data & OPCODE;
    i1 = LI | ( ( ( data & OPERND ) - 1 ) & OPERND );
    i2 = ADDI | 01;
    i3 = ST | ( addr & ADDRESS );
    if( badword( i1 ))
      {
        i1 = LI | ( ( ( data & OPERND ) - 2 ) & OPERND );
        i2 = ADDI | 02;
      }
    if( badword( ix ) || badword( i1 ) || badword( i2 ) )
      {
        printf( STD, "Cannot fix   %y:   %y*n",
             prthex, addr, prthex, data );
        exit();
      }

    if( badword( i3 ) )
      {
        i3 = ST;
        if( badword( i3 ) )
             error( "Cannot code an ST xxx (?xxx).*n" );
        j1 = LI | ( ( ( addr & ADDRESS ) - 1 ) & ADDRESS );
        j2 = ADDI | 01;
        j3 = ST | ( ( last + 6 ) & ADDRESS );
        if( badword( j1 ) || badword( j2 ) )
          {
            printf( STD, "Cannot fix   %y:   %y*n",
                 prthex, addr, prthex, data );
            exit();
          }
        while( badword( j3 ) )
          {
            count++;
            j3 = ST | ( ( last + 6 + count ) & ADDRESS );
          }
        code( j1 );
        code( j2 );
        code( j3 );
      }

    code( i1 );
    code( i2 );
    for( i = 1; i <= count; ++i )
         if( badword( NOP ) )
```

```
            error( "Cannot code a NOP (5000).*n" );
        else
            code( NOP );
    code( i3 );
    core[addr] = ix;
  }


badword( word )
  {
    if( badbyte( ( word & HIBYTE ) >> 8 ) || badbyte( word & LOBYTE ) )
        return( TRUE );
    else
        return( FALSE );
  }

badbyte( byte )
  {
    extrn newof, badch;
    auto i;

    for( i = 1; i <= badch[0]; ++i )
        if( ( byte + newof ) & LOBYTE == badch[i] )
            return( TRUE  );

    return( FALSE );
  }
```

photon/src/c/send.c

```
#include <stdio.h>
#define SIZE    4096      /* Number of words in Microdata core. */

/*
 * The output of `pho' is a unix core image (a.out).  This
 * consists of 16 bytes of info followed by a stream of bytes
 * which is the assembled code.  The code is followed by some
 * loader information which is ignored.
 */

    struct header
      {
        int h_magic;          /* Magic number */
        int h_tsize;          /* Size of program text segment */
        int h_stuff[6];       /* Six more words */
      } header;

int core[SIZE];           /* Core image for Photon */
char *coreb;              /* To reference core by bytes */
int f;                    /* Pointer to next unused word */

main()
  {
    coreb = core;
    f = ( cread() + 1 ) >> 1;
    cwrite();
  }

/*
 * Read the file into our core image.
 */

cread()
  {
    register n;
    register char *p;

    p = &header;
    for( n = 0; n < sizeof header; ++n )
        *p++ = getchar();
    for( n = 0; n < header.h_tsize; n++ )
      {
        if( n == 2 * SIZE )
            error( "Program too large for core\n" );
        coreb[n] = getchar();
      }
    return( n );
  }
```

```
/*
 * Write the core image out.
 */

cwrite()
  {
     register i, n;
     register char c;

     n = f << 1;
     for( i = 0; i < n; ++i )
       {
          if( ( i % 50 ) == 49 )
               putchar( '\n' );
          c = coreb[i];
          putchar( ( ( c >> 4 ) & 017 ) + 'a' );
          putchar( ( c & 017 ) + 'a' );
       }
     putchar( '\n' );
  }
```

photon/src/p/hcldr.p

```
/ This program consists of two parts.  The first part has two
/ manifests 'LOW' and 'HIGH'.  'LOW' is the address where
/ the second part begins, 'HIGH' is the address where the second
/ part is to be transferred to.  It is assumed that the user cal-
/ culates this high address so that the last word of the second
/ part is moved to address FFF.  After the second part has been
/ moved control is passed to the address given by the manifest
/ 'HFIRST'.  The second part is the high core loader.  It sets
/ up its interrupt address (storing it at location 1) and begins
/ to read bytes from the interface, constructing words and stor-
/ ing them in successive locations starting at 0 (with care taken
/ not to destroy the interrupt address).  When the 'EOB' code is
/ encountered a HALT is executed to stop the interrupts.  If the
/ user then hits RESTART (not RESET!) the words that should be in
/ locations 0 and 1 are loaded and control is tranferred to loc-
/ ation 0.  The program now in core is guaranteed to be the image
/ of the file transferred because the loader knows which bytes
/ should be loaded.  The user will have to ensure that these bytes
/ are never sent and that he does not try to load a program that
/ would overwrite the high core loader.
/ N.B.  This loader assumes that the first instruction of the program
/       being loaded is a jump (Axxx) and the second instruction is
/       a jump to subroutine (2xxx).

LOW     = 0x014 / Loader code starts at this address.
HIGH    = 0xfaf / Loader code will start at this address. (after move)
HFIRST  = 0xfe4 / Address of first instruction of transferred code.

HALTAD  = 0xfc9 / Where to go after boot is finished.
EOB     = 240   / Indicates End Of Boot when read by the second part.
DOFF    = 3     / All bytes read must have this subtracted.
INCR    = EOB - DOFF

/ Since the second part is going to be relocated all the addresses
/ calculated by 'pho' will be incorrect.  This must be corrected
/ by the user in the following way.  Each labelled address has a
/ manifest associated with it (the same name in upper case) which
/ has the value of the location which 'pho' should assign to the
/ symbol if relocation was possible.  Thus the following manifests.

STINT   = 0xfaf
INTER   = 0xfb0
SAVEAC  = 0xfbd
GET12   = 0xfbe
TEMP    = 0xfc5
WRAP    = 0xfc6
READ    = 0xfd4
AGAIN   = 0xfd5
NEXT    = 0xfef
STHGH   = 0xff6
```

```
STLOW   = 0xff9
INTADR  = 0xffb
DATA0   = 0xffc
DATA1   = 0xffd
ADDR    = 0xffe
REDTIM  = 0xfff

/ This is the first part that transfers the loader code
/ into high core.  It stops when it moves a word into
/ the location FFF.

        br   start    / First instruction must be a branch to allow fixes.

start:
        ld   low      / Address of next word to be moved.
        st   1f       / Save it away so we can load both the high...
        st   3f       / and low bytes separately.
        ld   high     / Where the word is to be moved to.
        st   2f       / Save it away so we can load both the high...
        st   4f       / and low bytes separately.

        refu          / Allow referencing the high 8 bits.
1:      ldb ..        / Get the high part of the word.
2:      stb ..        / Store the high part.
        refl          / Allow referencing the low 8 bits.
3:      ldb ..        / Get the low part of the word.
4:      stb ..        / Store the low part.

        isk low       / Address of next word to be moved.
        nop           / Drop through...
        isk high      / Where the next word is to be moved.
        br   start    / If it is not 0 then move the next word.
        br   HFIRST   / Else start executing high core loader.
low:         LOW
high:        HIGH

/ This is the second part which actually performs the loading.

/ The Microdata is interrupted every millisecond.  This interrupt
/ routine simply decrements a counter (if it is non-zero) and
/ returns control to the interrupted routine.

stint:
        0x9100        / Used for return address.
interrupt:
        br   ..
        st   SAVEAC   / Save the routines accumulator.

        ld   $0x500   / ?
        ioc  0x801    / ?
```

```
    add $-0x200 / ?
    refl        / ?
    st  STINT   / Store address to return to.

    ld  REDTIM
    sbeq        / If counter is not zero...
    add $-1     / decrement it.
    st  REDTIM

    ld  SAVEAC  / Restore accumulator...
    br  STINT   / and return to interrupted routine.
saveac:
    0           / Temporary storage for accumulator.

/ This routine reads two bytes from the interface and returns
/ in the AC a 12 bit quantity whose top 4 bits are the least
/ significant 4 bits of the first byte read and the bottom 8
/ bits are the 8 bits of the second byte.

get12:
    br  ..
    jst READ    / Read a byte.
    als $8      / Shift top 4 bits out, bottom 4 to high 4.
    st  TEMP    / Save this for a moment.
    jst READ    / Read another byte.
    add TEMP    / Form the 12 bit quantity desired.
    br  GET12
temp:
    0

/ Control is transferred here when the 'EOB' code is encountered.
/ Location 0 is loaded with a tranfer back to the instruction
/ after the HALT just in case the user hits the switch RESET.
/ Note that we know that the high bits of location 0 are 1010 (A)
/ which indicates a branch.  Thus we only need load the low bits.
/ A HALT is executed to turn off interrupts, and when RESTART is
/ pressed locations 0 and 1 of the program loaded are fixed.  This
/ could not be done with interrupts on.  Control is then passed to
/ location 0 and the loaded program starts executing (usually the
/ patches need to have it read by this loader.)

wrap:
    ld  $HALTAD / Set up location 0 so that we return to...
    st  0       / the right address even if RESET is hit.
    halt        / Effectively: turn off interrupts.
    refu
    ldb DATA0   / Load high byte of location 0...
    stb 0       / and store it.
    ldb DATA1   / Load high byte of location 1...
    stb 1       / and store it.
```

```
    refl
    ldb DATA0    / Load low byte of location 0...
    stb 0        / and store it.
    ldb DATA1    / Load low byte of location 1...
    stb 1        / and store it.
    br  0        / Transfer control to loaded program.

/ This routine reads a byte from the interface, checks to see
/ if it is the `EOB' code and removes the offset given by
/ `DOFF'.  It returns an 8 bit quantity in the AC.  If the
/ byte is `EOB' control is passed to the label `wrap'.

read:
    br  ..

again:
    ld  REDTIM   / Is there a read in progress?
    bne AGAIN    / Yes.  Wait until it is finished.

    ld  $3       / Three milliseconds for a read.
    st  REDTIM

    ioc 0x702    / Read next byte from interface.
    bbeq AGAIN   / Ignore nulls.
    add $-EOB    / Is it the end of the boot?
    sbne
    br  WRAP     / Yes.  Transfer control to wrapup.
    add $INCR    / No.  Remove offset.

    als $4
    ars $4       / Make sure we just return 8 bits.

    br  READ

/ This where control should be transferred when the move to high
/ core is complete.  First the interrupt address is put into loc-
/ ation 1 and interrupts are turned on.  The first two words are
/ read and saved for storage by the wrapup.  Then bytes are read
/ and stored in successive locations starting at address 2.

    refu
    ldb INTADR
    stb 1        / Store top 8 bits of interrupt address.
    refl
    ldb INTADR
    stb 1        / Store bottom 8 bits of interrupt address.
    0x9100       / Turn on interrupts.

    jst GET12
    st  DATA0    / Get and save operand for instruction at location 0.
```

```
    jst GET12
    st  DATA1    / Get and save operand for instruction at location 1.

next:
    ld  ADDR     / Last word stored.
    add $1
    st  ADDR     / Word to be stored.
    st  STHGH
    st  STLOW

    jst READ     / Read a byte.
    refu
sthgh:
    stb ..       / Store in top part of word.
    refl
    jst READ     / Read a byte.
stlow:
    stb ..       / Store in bottom part of word.
    br  NEXT     / Repeat.
intadr:
    0x2fb0       / Address of interrupt routine.
data0:
    0xa000       / Temporary storage for instruction at location 0.
data1:
    0x2000       / Temporary storage for instruction at location 1.
addr:
    1            / Last address to be stored.
redtim:
    0            / Time needed to finish current read.
```

```
/ This is the source for the boot file in ~jhbuccino/ph/orig.
/ It is also the same as ~photon/p/troff' except that the latter
/ has 4 extra words, all containing FFFF after the last halt
/ and before the trailing 0A00.
skp =   0 ^ ioc
ACOM    =   0x10          /Number of available commands
NCOM    =   10          /Number of valid commands
nop =   0x5000

/This program runs the Photon Econosetter.
/   Copyright (c) 1978, Johann H. George

LEDFWD  =   0x106
DRPPWL  =   0x140
FLASH   =   0x201
CARFWD  =   0x204
ROWS    =   0x210
ROW0    =   0x240
REDLNS  =   0x308
FLOP    =   0x402
CARBKD  =   0x404
ROW1    =   0x440
READ    =   0x702
PLSLED  =   0x804
LEDBKD  =   0x805
LFTPWL  =   0x840

PTPRDY  =   0x082
FLSCMP  =   0x130

    br  init          /Jump around interrupt
    jst interrupt     /Interrupt

/Initialization
/
init:
    ld  $0
    st  redtim
    st  carchg
    st  ledval
    st  ledchg
    st  ledtim
    st  dskoff
    st  widoff
    st  carval
    st  pntoff
    st  pntsiz
    st  fntoff

    ld  $3       /RESET sets us to inner row
```

```
	st   row

	ld   $135        /Software offset for all output
	st   carval
	neg
	st   carpos

	ld   $1
	st   dir

	ld   $0      /Change typeface to Roman
	jst movfac

	ld   $1      /Change to pointsize 10
	jst movpnt

	0x9100           /Turn on interrupts

/Read characters and branch accordingly
/    The characters from 0x00 - 0x1f are taken to be commands.
/    Most of these are unused.  The rest of the characters,
/    map onto a character which is flashed.
/
loop:
	jst read         /Read a character
	add $-ACOM       /See if a flashable character
	bge character    /Character
	add $ACOM-NCOM   /Valid command?
	sklt
	jst error        /Illegal command
	add $jmptab      /Form address for jump table
	add $NCOM
	st   0f      /Stuff jump
0:
	br   ..      /And off we go

jmptab:              /Jump table
	br   loop        /Ignore nulls
	br   setfac      /Change typeface
	br   setdir      /Change direction
	br   setpnt      /Change pointsize
	br   setcar      /Move carriage
	br   setled      /Leading
	br   reset       /Set carriage position to zero
	br   setcol      /Change current colour
	br   pgmrk       /Start new page
	br   done        /Finished typesetting

/
1:               /Part of code for return from interrupt
```

```
    0x9100          /Enable interrupts after 3 instructions
interrupt:          /Routine really starts here
    br  ..      /Return
    st  9f      /Save accumulator
    ld  $0x500      /Set reference on return?
    ioc 0x801       /?
    add $-0x200
    refl
    st  1b

    jst movcar      /Move carriage if needed.
    ld  ledtim      /See if previous leading has finished
    sbeq            /Decrement timer
    add $-1
    st  ledtim
    sbne
    jst movled      /Lead if needed

    ld  redtim      /Decrement reader timer
    sbeq
    add $-1
    st  redtim

    ld  9f      /Restore accumulator
    br  1b      /Return

9:
    0

/Move carriage
/
movcar:
    br  ..
    ld  carchg      /In the middle of changing carval?
    bne movcar      /Yes.  What a pity

    ld  carval      /Want to move carriage?
    blt 3f      /Move backwards
    bne 2f      /Move forwards
    br  movcar      /No move


2:
    add $-1     /Update carval as we move
    st  carval

    ioc CARFWD      /Move carriage forward
    br  movcar      /At limit switch

    ld  $32     /Wait for move to complete
```

```
1:
	add $-1
	bne 1b

	ioc CARFWD	/Move again
	br  movcar	/At limit switch

	ld  carpos	/Update carriage position
	add $1
	st  carpos
	br  movcar

3:
	add $1	/Update distance to move
	st  carval

	ioc CARBKD	/Move carriage backwards
	ld  $32	/Wait to complete
1:
	add $-1
	bne 1b
	ioc CARBKD	/Move again

	ld  carpos	/Update carriage position
	add $-1
	st  carpos
	br  movcar

/Lead
/
movled:
	br  ..
	ld  ledchg	/In the middle of changing ledval?
	bne movled	/Bad time to catch us

	ld  ledval	/Distance to move
	blt 2f	/Move backwards
	bne 1f	/Move forwards
	br  movled	/No move

1:
	add $-1	/Update ledval as we move
	st  ledval
	ioc LEDFWD	/Set lead switch forward
	br  3f
2:
	add $1	/Update ledval
	st  ledval
	ioc LEDBKD	/Set lead switch backward
3:
```

```
	ioc PLSLED	/Pulse forward
	ld  $11		/Takes 11 miliseconds to pulse
	st  ledtim
	br  movled

/Flash a character.
/	If dir is positive, we first move the width of the
/	character.  If dir is negative, we flash the character
/	and then move its width backwards.  If dir is zero, we
/	don't move at all.
/	I should worry about multiflash.
/
character:
	st  9f		/Save char away
	ld  dir		/direction
	bblt	2f
	bbeq	1f
	ld  9f
	jst lookwd	/Lookup width
	jst addcar	/Move width of char
1:
	ld  9f		/Get char and flash
	jst flash
	br  3f		/Done
2:
	ld  9f		/Get char and flash
	jst flash
	ld  9f
	jst lookwd	/Lookup width
	neg		/We want to move backwards
	jst addcar
3:
	br  loop

9:
	0		/Temp storage for char

/Change typeface
/
setfac:
	jst read	/Get new typeface
	jst movfac
	br  loop

/Change typeface
/	The following must be accomplished.
/	1) The offset into the width table must be changed tc
/	   correspond to the new typeface.
/	2) The offset used when flashing a character must be
/	   changed as odd typefaces are on the same row as
```

```
/       the previous even typeface, except 112 characters
/       ahead.
/    3) We have to select which row of typefaces we want.
/
movfac:
     br   ..
     st   9f        /And save

     ror $1
     blt 1f
     ld   $0        /Even typeface
     br   2f
1:
     ld   $112          /Odd typeface
2:
     st   widoff
     st   dskoff


     ld   row       /Reset current row
     st   8f
     ld   9f
     ars $1
     st   row

     beq 1f
     ioc ROW1           /Move shutter to position?
     ld   widoff
     add $2*112
     st   widoff
     br   2f
1:
     ioc ROW0           /Move shutter to position?
2:
     ld   row       /Calculate distance to move carriage
     neg
     add 8f
     als $6         /64 per font
     st   7f
     jst addcar         /Move carriage

     ld   fntoff
     add 7f
     st   fntoff

     ld   $16       /Delay for a while
     jst delay
     br   movfac

8:
```

```
    0           /Old row
9:
    0           /Storage for typeface

/Change pointsize
/
setpnt:
    jst read        /Get new pointsize
    jst movpnt      /Change pointsize
    br  loop

/Change pointsize
/   We are passed an offset into the table pstab
/   which contains a list of pointsizes.
/
movpnt:
    br  ..
    st  6f      /Index for new pointsize

    ld  pntsiz      /Old pointsize
    st  8f
    ld  pntoff      /Old carriage offset for pointsize
    st  7f

    ld  6f      /Find new pointsize from index
    add $psztab     /Pointsize table
    st  1f
1:
    ld  ..
    st  pntsiz

    ld  6f      /Get new lense code and save complement
    add $lnctab
    st  1f
1:
    ld  ..
    neg
    st  9f

    ld  6f      /Find new carriage offset
    add $poftab     /Offset table
    st  1f
1:
    ld  ..
    st  pntoff

    isk carchg      /About to change carval
    nop

    ld  8f      /pos =+ (pos*old)/new - pos
```

```
    st   mult         /Multiplicand

    ld   7f      /Position carriage should be before change
    add fntoff        /Subtract offset for font as it is constant
    neg
    add carpos
    add carval
    st   6f      /Save pos for subtracting later
    jst multiply

    ld   pntsiz
    jst divide

    neg
    add 6f       /Subtract pos
    add 7f       /Subtract old offset
    neg

    add carval        /Hadn't moved this yet
    add pntoff        /New offset

    st   carval
    ld   $0
    st   carchg

    ioc REDLNS        /Are we already where we want to be?
    add 9f
    beq movpnt

    ioc LFTPWL        /Initiate lense motion
1:
    ioc REDLNS        /Read lense code
    st   6f

    ld   $1      /Delay for a while
    jst delay

    ioc REDLNS        /We don't believe last result, so repeat
    neg
    add 6f
    bne 1b       /They weren't the same.  Try again

    ld   6f      /Are we now where we want to be?
    add 9f
    bne 1b       /No.  Keep trying

    ioc DRPPWL        /Drop pawl
    ld   $0x80        /Delay for a while
    jst delay
    ioc REDLNS        /See if we settled in the right place
```

```
    add 9f
    bne 1b          /We missed

    br  movpnt

6:
    0               /Temporary
7:
    0               /Old carriage offset for pointsize
8:
    0               /Old point size
9:
    0               /Lense code for new pointsize

lnctab:
    0x000; 0x200; 0x300; 0x100
psztab:
    8; 10; 14; 18
poftab:
    1; 205; 438; 559

/Change direction
/   Change the direction we are moving.  Look under the
/   code for char.
/
setdir:
    jst read        /Get new direction
    jst sgnext      /Extend sign from 7 bits to 12
    st  dir     /And save
    br  loop

/Move the carriage.
/
setcar:
    jst read        /Amount to move carriage
    jst sgnext      /Sign extend
    jst addcar      /Move carriage
    br  loop

/The given value is added to the current distance the
/carriage has to be moved (carval).
/
addcar:
    br  ..
    isk carchg      /Let int handler know carval is being changed
    nop
    add carval
    st  carval
    ld  $0      /Finished changing carval
    st  carchg
```

```
    br  addcar

/Leading
/
setled:
    jst read        /Amount to lead
    jst sgnext      /Sign extend
    isk ledchg      /We are changing ledchg
    nop
    add ledval
    st  ledval
    ld  $0      /Done changing ledval
    st  ledchg
    br  loop

/Set the carriage position to zero
/
reset:
    isk carchg      /Let int handler know carval is being changed
    nop

    ld  carpos      /Get current carriage position
    neg         /We want to go back to zero
    add pntoff      /Add offset for pointsize
    add fntoff      /Carriage offset
    st  carval      /Store for carriage position

    ld  $0      /Done changing carval
    st  carchg
    br  loop

/Flash the given char
/   The carriage might have to move one more step?
/   I should worry about multiflash.
/
flash:
    br  ..

    st  9f      /Save character

1:
    ld  ledtim      /Wait for leading to complete
    bne 1b

1:
    ld  carval      /Wait for carriage to finish moving
    bne 1b

    ioc ROWS        /I don't know what this does
    ld  9f      /Get char
```

```
	add dskoff
	add $1		/Its origin is one
	ioc FLASH		/Flash
1:
	skp FLSCMP		/Wait for flash to complete
	br  1b
	br  flash
9:
	0		/For character

/Read a character
/	We only want the bottom seven bits.
/
read:
	br  ..
1:
	ld  redtim		/See if we can read a character
	bne 1b

	ld  $3		/Three miliseconds for read to complete
	st  redtim

	ioc READ		/Read a char via the interface
	bbeq	1b		/Skip nulls

	als $5		/Get rid of top bit
	ars $5
	br  read		/Return

/Multiply the value passed in the accumulator by that in mult.
/	The accumulator is a 12 bit value, and mult is an 11 bit
/	value.  The bottom 11 bits of the result is stored in d2.
/	The high order bits are stored in d1.
/
8:
multiply:
	br  ..
	st  9f

	ld  $0		/Initialize result
	st  d1
	st  d2

	ld  9f
	bge 1f		/Multiply by the 12th bit and truncate to 11
	add $0x800		/Get rid of top bit
	st  9f
	ld  mult
	st  d1
1:
```

```
	ld   mult		/See if done
	beq 3f

	ld   d2		/Add with carry
	add 9f
	st   d2
	bge 2f

	add $0x800		/Get rid of top bit
	st   d2
	ld   d1		/And increment upper half
	add $1
	st   d1

2:
	ld   mult		/Decrement and see if done
	add $-1
	st   mult
	br   1b
3:
	ld   d2		/Lower half of result
/	br   multiply
	br   8b		/Previous statement gives error

9:
	0		/Storage for other multiplier
mult:
	0		/First multiplier
d1:
	0		/Result for multiply or dividend for divide
d2:
	0

/Divide the 22 bit integer d1, d2 (the bottom 11 bits are used)
/by the 11 bit value passed in the accumulator.  d1 and d2 are
/destroyed.
/
divide:
	br   ..
	neg
	st   9f		/Save negative of divisor

	ld   $0		/Result
	st   8f

	ld   $0x800		/Bit position in result
	st   7f

2:
	ld   d1		/See if result contains this bit
```

```
        add 9f
        blt 1f

        st  d1          /Save new value
        ld  7f          /Add bit to result
        add 8f
        st  8f

1:
        ld  7f          /Shift bit right one position
        ars $1
        st  7f
        beq 3f          /Done

        ld  d1          /Shift d1,d2 left one position
        als $1
        st  d1

        ld  d2
        als $1
        st  d2

        ars $11
        add d1
        st  d1

        br  2b

3:
        ld  8f
        br  divide

7:
        0               /Current bit position in result
8:
        0               /Result
9:
        0               /Negative of divisor

/Extend sign from an seven bit number to a twelve bit number.
/
sgnext:
        br  ..
        als $5          /Get sign bit to top of register
        bge 1f          /Positive
        add $0x1f           /Extend sign bit
1:
        ror $5          /Move back
        br  sgnext
```

```
/Look up width of character passed
/
lookwd:
    br  ..
    add widoff      /Add offset (depending of typeface)
    ror $1      /See if width is in lower or upper
    refu            /Assume width is in upper half
    skge
    refl            /Guilty
    als $1      /Word offset into width table
    ars $1
    add $widtab     /Form address for lookup
    st  1f
1:
    ldb ..      /Finally look up width
    refl            /Set to lower before we forget
    br  lookwd      /Return

/Delay for quite a while
/
delay:
    br  ..
    st  9f

2:
    ld  $0x100      /Short delay
1:
    add $-1
    bne 1b

    ld  9f      /See if done
    add $-1
    st  9f
    bne 2b

    jmp delay

9:
    0           /Countdown timer for delay

/Error routine
/   Halt
/
error:
    br  ..      /Will contain return address
    st  9f      /Save ac
    ld  error       /We load return address int ac
    halt            /And die
9:
    0           /Save for ac
```

```
/ Flushes the change colour command along with the following
/ byte that selects the colour.

setcol:
    jst read        /Read colour number
    br  loop

/ Flushes the page mark command as well as the following
/ two bytes which contain the page number.

pgmrk:
    jst read
    jst read
    br  loop

/ We have received the end of document character from Honeywell.
/ We have to make sure that all carriage movement and leading
/ have completed.  We then halt the Microdata.

done:
1:
    ld  ledval
    bne 1b          /Make sure all leading is completed
1:
    ld  carval
    bne 1b          /Make sure carriage has finished moving
1:
    halt
    br  1b          /Make sure user hits RESET

redtim:
    0               /Timer for read to complete
carval:
    0               /Distance carriage should move
carchg:
    0               /Carval is being changed
carpos:
    0               /Current position of carriage
ledval:
    0               /Amount left to lead
ledchg:
    0               /Ledval is being changed
ledtim:
    0               /Countdown timer while waiting for lead
dir:
    1               /Current direction we are moving
dskoff:
    0               /Offset on disk when flashing
widoff:
    0               /Offset into width table
```

```
pntsiz:
    10              /Current pointsize
pntoff:
    0               /Offset from margin for pointsize
fntoff:
    0               /Offset carriage moves for a font
row:
    0               /Current row on disk

/Width table
/   Character 407 used to have a width of 21
/
widtab:
    .byte   18, 18, 18, 18, 18, 18, 18, 18
    .byte   18, 18, 18, 12, 12, 16, 12, 18
    .byte   10, 19, 14, 19, 14, 19, 19, 10
    .byte   19, 16, 12, 30, 18, 25, 19, 18
    .byte   19, 17, 20, 19, 10, 18, 16, 10
    .byte   10, 10, 10, 18, 12, 10, 18, 36
    .byte   28, 28, 28, 28, 22, 36, 18, 36
    .byte   30, 14, 12, 12, 30, 24, 24, 28
    .byte   15, 30, 22, 28, 28, 30, 28, 24
    .byte   30, 26, 23, 36, 28, 36, 23, 28
    .byte   24, 26, 28, 28, 17, 28, 25, 10
    .byte   10, 10, 10, 18, 12, 14, 18, 10
    .byte   20, 20, 22, 18, 30, 30, 30, 24
    .byte   14, 30, 18, 18, 30, 18, 18, 30
    .byte   18, 18, 18, 18, 18, 18, 18, 18
    .byte   18, 18, 18, 12, 12, 16, 12, 18
    .byte   10, 19, 14, 19, 14, 19, 19, 10
    .byte   19, 16, 12, 30, 18, 25, 19, 18
    .byte   19, 17, 20, 19, 10, 18, 16, 10
    .byte   10, 10, 10, 18, 12, 10, 18, 36
    .byte   28, 28, 28, 28, 22, 36, 18, 36
    .byte   30, 14, 12, 12, 30, 24, 24, 28
    .byte   15, 30, 22, 28, 28, 30, 28, 24
    .byte   30, 26, 23, 36, 28, 36, 23, 28
    .byte   24, 26, 28, 28, 17, 28, 25, 10
    .byte   10, 10, 10, 18, 12, 14, 18, 10
    .byte   20, 20, 22, 18, 30, 30, 30, 24
    .byte   20, 28, 18, 18, 30, 18, 18, 30
    .byte   18, 18, 18, 18, 18, 18, 18, 18
    .byte   18, 18, 18, 12, 12, 16, 12, 18
    .byte   10, 18, 14, 18, 14, 18, 18, 10
    .byte   18, 16, 12, 28, 18, 24, 18, 18
    .byte   18, 14, 20, 18, 10, 20, 16, 10
    .byte   10, 12, 10, 18, 12, 10, 18, 36
    .byte   28, 28, 28, 28, 22, 36, 18, 36
    .byte   30, 14, 12, 12, 30, 24, 24, 26
    .byte   14, 28, 22, 28, 26, 30, 28, 24
```

```
.byte   28, 26, 22, 36, 26, 36, 24, 26
.byte   24, 26, 26, 28, 18, 26, 26, 10
.byte   10, 12, 10, 18, 12, 14, 18, 10
.byte   20, 20, 22, 16, 30, 30, 30, 24
.byte   20, 22, 18, 18, 30, 18, 18, 30
.byte   36, 36, 14, 14, 30, 18, 30, 30
.byte   30, 30, 30, 30, 30, 22, 14, 16
.byte   18, 22, 20, 16, 16, 22, 20, 14
.byte   16, 22, 16, 16, 18, 20, 12, 16
.byte   22, 22, 20, 22, 18, 16, 18, 30
.byte   30, 18, 18, 30, 12, 36, 18, 10
.byte   18, 18, 18, 18, 18, 18, 18, 18
.byte   18, 18, 30, 30, 18, 26, 26, 24
.byte   24, 36, 30, 26, 26, 30, 18, 34
.byte   26, 18, 14, 16, 30, 26, 30, 36
.byte   26, 36, 36, 24, 30, 30, 30, 30
.byte   30, 18, 30, 30, 12, 18, 22, 10
.byte   18, 36, 36, 18, 30, 30, 14, 16
.byte   16, 16, 30, 16, 30, 30, 30, 18
```

Appendix E

The Microdata

This appendix contains useful information on programming the Microdata. The instruction set is given, then the use of the front panel switches is explained. Finally, the use of the assembly language PHO is described.

Instruction Set E.1.1
Front Panel E.1.3
Fetch & Store E.1.4
The PHO Language

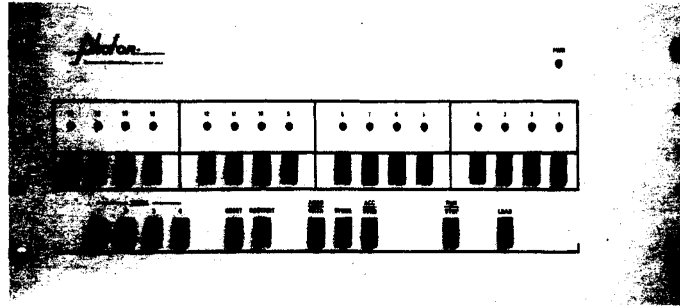
Instruction set

HEX	Mnem.	Description
0000	NOP	- Six microsecond delay
0001	SKP	- Skip when EOL flip-flop is not set
0002	SKP	- Skip if keyboard is on line
0004	SKP	- Skip if MAN/AUTO LINE switch is in MAN
0008	SLZ	- Skip if accumulator bits 1-8 equal zero
0010	SLP	- Skip if accumulator bit 8 is zero
0020	SLM	- Skip if accumulator bit 8 is a one
0038	JMP	- Unconditional skip
0070	SLNZ	- Skip if accumulator bits 1-8 do not equal zero
0130	SKPFL	- Skip when flash complete
0208	SKZ	- Skip if accumulator bits 1-12 equal zero
0230	SHZ	- Skip if accumulator bits 9-12 equal zero
0430	SKP	- Skip if accumulator bit 12 is a zero
0830	SKM	- Skip if accumulator bit 12 is a one
1401	CLONE	- Invert data in accumulator
1500	INCA	- Add one to the accumulator
1501	CTWO	- Invert data in accumulator and add one
1505	DECA	- Subtract one from accumulator
18XX	ALS	- Shift accumulator left XX times
1AXX	ROL	- Rotate accumulator left XX times
1CXX	SRT	- Shift accumulator right XX times
1EXX	ROR	- Rotate accumulator right XX times (number of rotations or shifts equals amount of ones in low eight bits, for example 07 equals 3 shifts)
2NNN	JST	- Store program counter in address NNN and change program counter to NNN+1
3XXX	LI	- Clear accumulator with XXX
4NNN	ISK	- Increment bits 1-8 (9200) or (9-16) of address NNN and skip next instruction if all bits are zero after the increment
5XXX	ADDI	- Add XXX to the accumulator
6NNN	LOAD	- Clear accumulator and load accumulator with bits 1-8 (9200) or 9-16 (9400) from location NNN
7NNN	STA	- Store accumulator bits 1-12 in address NNN
8NNN	ADD	- Add bits 1-8 (9200) or 9-16 (9400) from address NNN to accumulator
9100	HEX	- Allow interrupts after three more instructions
9200	REFL	- Enable circuits for an eight bit addition or storage instruction into address bits 1-8
9400	REFU	- Enable circuits for an eight bit addition or storage instruction into address bits 9-16
9800	HALT	- Stop the master clock
ANNN	JMP	- Change program counter to NNN

Code	Mnemonic	Description
B000	NOP	- Six microsecond delay
CNNN	STO	- Store accumulator bits 1-8 ([illegible]200) or 9-16 (9400) in address NNN
DNNN	ADD12	- Add bits 1-12 in address NNN to accumulator
E000	IOC	- Skip when OSL lamp on keyboard is illuminated
E101	IOC	- Turn on Min Space lamp on keyboard
E106	IOC	- Pulse leading motor forward
E140	IOC	- Stop lens motion
E201	IOC	- Flash character
E204	IOC	- Carriage forward
E208	IOC	- Input lens encoder
E210	IOC	- Select row 1 (typefaces 1 and 2)
E220	IOC	- Input leading switches
E240	IOC	- Enable lens relay for reverse operation
E280	IOC	- Skip if MAN/AUTO LEAD switch is in AUTO
E302	IOC	- Clear accumulator, input reader frame, but do not step reader
E308	IOC	- Clear accumulator and input lens encoder
E320	IOC	- Clear accumulator and input leading switches
E401	IOC	- Set EOL flip-flop
E404	IOC	- Carriage reverse
E408	IOC	- Skip if MAN/AUTO LENS switch is in AUTO
E420	IOC	- Input lens position switch and add to accumulator
E440	IOC	- Enable lens relay for forward operation
E480	IOC	- Skip if LOAD/NORM switch is in LOAD positin
E520	IOC	- Input lens position switch and clear accumulator
E602	IOC	- Input reader and step one frame
E610	IOC	- Select row 2 (typefaces 3 and 4)
E702	IOC	- Clear accumulator, input reader frame and step reader
E801	IOC	- Skip if bits 9-16 are high
E802	IOC	- Skip if HYPH/NORM is in HYPH position
E804	IOC	- Step leading motor
E805	IOC	- Pulse leading motor reverse
E840	IOC	- Initiate lens motion
E880	IOC	- Turn on OSL on keyboard
E908	IOC	- Input right side line length switches
E920	IOC	- Input left side line length switches
EA10	IOC	- Select row 3 (typefaces 5 and 6)
EA40	IOC	- Enable lens relay for reverse and initiate lens motion
EC10	IOC	- Select row 4 (typefaces 7 and 8)
EC40	IOC	- Enable lens relay for forward and initiate lens motion
FNNN	LDD	- Clear accumulator and load accumulator with bits 1-12 from address NNN

Front Panel

Figure E.2.1 shows the different controls located on the front panel of the Microdata. An description for the use of these switches is given below.



SENSE 1 - Allows the machine to be stpped on selected addresses prior to the execution of instructions which modify memory.

SENSE 2 - Allows the machine to be stpooed when processing data from selected addresses

SENSE 3 - Enables front panel indicators to illuminate when machine is operating

SENSE 4 - Used to select an address from the panel switches

Switches 1-16 - Used for selecting memory addresses and changing data in memory

RESET - Resets or initializes all logical functions and clears all registers

RESTART - Starts or initiates system clock

STORE - Allows data selected by switches 1-16 to be entered into memory

RUN/STOP - Performs a machine HALT operation in the HALT position

LOAD - Performs a hardware fill memory operation

ACC/STRB - Allows accumulator to be displayed on indicators 1-16

ADDR/DATA - Selects memory address in ADDR position or memory contents in DATA position for display on the front panel indicator lights

Fetch Procedure

1. Set all toggles on the Microdata panel to their 'up' position, except for the SENSE 3 switch which must be in the down position.
2. Press and release the RESET switch.
3. Set the RUN/STOP switch to the STOP position.
4. Set the sixteen data switches to the hexadecimal address of the desired location. (up=0, down=1)
5. Set the SENSE 4 switch to its down position.
6. Press and release the RESTART switch twice.
7. To display data set the ADDR/DATA switch to the DATA position and observe the indicators. To display the address set the switch to the ADDR position.
8. To display the contents of the accumulator press the ACC/STRB switch and observe the indicators.
9. The memory address is incrememted by 1 every time the RESTART switch is depressed after step 7. In this way successive locations can be examined. To continue, return to step 6.

Store Procedure

1. Set all toggles on the Microdata panel to their 'up' position, except for the SENSE 3 switch which must be in the down position.
2. Press and release the RESET switch.
3. Set the RUN/STOP switch to the STOP position.
4. Set the sixteen data switches to the hexadecimal address of the desired location. (up=0, down=1)
5. Set the SENSE 4 switch to its down position.
6. Press and release the RESTART switch twice.
7. To display data set the ADDR/DATA switch to the DATA position and observe the indicators. To display the address set the switch to the ADDR position.
8. To display the contents of the accumulator press the ACC/STRB switch and observe the indicators.
9. Enter the data to be stored using the data switches.
10. Press and release the STORE switch. Data is now stored in the current location and the current location is incremented by 1. Successive locations can be stored by repeated pressing of the STORE switch. To continue, return to step 7.

The Photon Assembly Language

Charles H. Forsyth

University of Waterloo,
Waterloo, Ontario N2L 3G1

1. Introduction

The Photon assembler, *pho*, is a two-pass assembler written in C and YACC, which runs on UNIX and assembles a source programme into a UNIX-style object module. If there are no undefined external references, the object module may then be downloaded, and executed, without further processing; otherwise, it may be combined with other object files by a link editor.

The assembler supports no macro facility, but another command such as the C pre-processor, *m4*, *ml1*, or even *nroff* might be pressed into service as a macro pre-processor. The syntax of the assembler owes much to that of *as*, the UNIX assembler for the PDP-11, and is rather different from that of the manufacturer's assembler. Names of operation codes, and the format of operands or addresses are also often different. This document describes the use of the assembler, the syntax of an assembly-language programme, and the format of the available operations, noting differences with the manufacturer's assembler, where appropriate.

2. Usage

The assembler is invoked by the command line:

pho [**−**] [*name* ...]

The named source files are concatenated, and assembled, and the resulting object module is left in the file **a.out** in the current directory. If no source files are supplied, the standard input is read; thus *pho* may be used at the end of a pipeline.

If the optional **−** flag is given, all names which are undefined at the end of the assembly are made undefined-external.

3. Format, and Basic Elements of the Source Programme

An assembly-language programme is a sequence of statements, separated by newlines, or semi-colons, the latter allowing several statements per line. A statement is typically a machine instruction, with appropriate operands, but may be an assembly-time expression, or an assembler pseudo-operation.

The assembler accepts free-format input — it knows nothing of columns or sequence-number fields, and blanks and tabs may be used freely between tokens. A comment may be started anywhere on a line by a slash '/', which causes the characters following, to the end of the line, to be ignored.

3.1. Names

Names may be of any length, although only the first eight characters are significant. They may be formed from the letters 'a' through 'z', 'A' through 'Z', the underscore '_', the period '.', and the digits '0' through '9'. The first character may not be numeric, and upper- and lower-case letters are considered distinct.

3.1.1. The Location Counter

The location counter has a legal name, like any other symbol, and may therefore be used in any expression where a name might be valid. This name is '.', and it has the type relocatable; its value is the address, within the current segment of the programme, of the next byte to be written.

3.1.2. The Relocation Value

The symbol '..' holds the current relocation value for the programme, which starts off as zero. The relocation value is added to all text, data, or bss segment references, and is subtracted from all pc-relative references to absolute addresses, if the hardware provides such an addressing mode. (The Photon does not.)

The relocation value is provided for those times that a piece of code is assembled at some address, but will be moved to another location before it runs. For relocation of an entire programme, the *reloc* command is usually better, as it may more safely be used with the link editor.

3.2. Labels

A label definition may appear before any statement (including the empty statement), and consists of a name, followed by a colon. In effect, this causes the current value and type of '.' to be assigned to the name. It is required that the value assigned be the same in the second pass, as in the first. Violation of this results in the diagnostic "phase error".

3.3. Temporary Labels

The digits '0' through '9' may each be associated with statements in the programme, in much the way that labels are. These *temporary labels* differ in that any of them may appear as label to many different statements. Reference to a temporary label is made by writing either the letter **f** or **b** following the label number, indicating the first such label in a forward or backward direction from the reference.

3.4. Constants

3.4.1. Numbers

The sequence '0x' (or '0X') announces a hexadecimal constant, which consists of a sequence of the digits '0' to '9', or the letters 'a' through 'f' (or 'A' through 'F'). A sequence of digits beginning with '0' identifies an octal constant; the digits '8' and '9' are the octal digits '10' and '11'. Any other sequence of digits forms a decimal constant.

3.4.2. Characters

An apostrophe ‘´’ followed by up to two ASCII characters, followed by a closing apostrophe, forms a 16-bit word with the characters right-justified, and padded on the left with zeroes. An escape sequence is recognised inside the character constant to allow for the entry of otherwise inconvenient characters, namely:

\" produces "
\´ ´
\\ \
\0 NULL (000)
\e EOT (004)
\n newline (012)
\r carriage return (015)
\f form feed (014)
\t tab character (011)
\nnn octal byte ‘nnn’

3.5. Segments

There are three segments in a programme: text, data, and bss. The text segment is expected to contain most of the code, and constant data. The data segment provides working storage, and may also house impure sections of code. The bss segment may contain storage reservation operations only, and is guaranteed to be zero at the start of execution; the assembler may then arrange that this space is not taken up in the object file.

Initially, the value of ‘.’ within each segment is zero, and ‘.’ is placed in the text segment. Before a new segment is entered, the current value of ‘.’ within the old segment is saved, to be restored when that segment is re-entered.

4. Expressions

Expressions are formed, in the usual manner, from operands of various types, which are combined by a number of binary operators. The terms of an expression are names, including labels and temporary labels, and constants.

4.1. Operators

All operators are binary operators, and have equal precedence. Most operators use a zero value of absolute type if the left operand is missing; this provides the usual unary operators. The exception is the operator ‘*’, the unary use of which is reserved for use in constructing addresses. The complete set of operators is:

+ addition
- subtraction
* multiplication
\/ division (‘/’ by itself is a comment)

% modulus

& bit-wise and

| bit-wise or

! bit-wise or of the first operand with the one's complement of the second; usually used as a unary operator to obtain the one's complement

<< logical left shift of the first operand by the number of bits given by the second operand

>> corresponding logical right shift

^ take the value of the first operand, and the type of the second; used to assign absolute values to '.' without changing its type, and to define new machine instructions with the syntax of an existing one.

Expressions are evaluated from left-to-right except that sub-expressions may be surrounded by brackets "[]" to override the usual precedence.

4.2. Types of Values

Each term and expression has a type, which says something about its relocation properties, and identifies special symbols to the assembler.

The basic types, and their properties, are:

undefined — Names which are new to the assembler in the first pass are of type undefined. This type is changed for another by assignment, or when the name is used as a label. Pass 1 generally tolerates undefined symbols (to allow forward references), but it is an error to refer to an undefined symbol in Pass 2.

absolute — All constants have absolute type, as do names defined from them. This means that references to them are not subject to relocation by the loader.

relocatable — Symbols defined in any of the text, data, or bss segments have the type of that segment, and the attribute *relocatable*. Thus, a text symbol and a bss symbol are not of the same type, but both may be subject to relocation by the link editor, if the object module is combined with others.

external — Names which have one of the types absolute, text, data, or bss may appear in a **.globl** pseudo-operation, which causes them to acquire the "external" attribute. Such names act as their non-external counterparts do, but are made known to the link editor, and may therefore be referenced by other object modules.

Names which appear in a **.globl** statement, but are not otherwise defined in the programme have the type undefined-external, and must eventually be defined

with the aid of the link editor.

keyword — A name with a keyword type is usually a machine operation, or an assembler pseudo-operation, where the type of keyword identifies the syntax of the operation, and the value, the particular operation.

Most operators may only be applied to expressions of type absolute, or absolute-external, although in Pass 1 if an operand to any operator is undefined, the operation is legal, and the type of the expression is undefined. When combined with others, the non-relocatable types are treated as absolute; the type of the result, of the two operand types, will be that which appears latest in the above table, when any external properties are disregarded.

The additive operators widen the types of operands accepted to include relocatable types, with certain restrictions. A relocatable value may have an absolute offset value added or subtracted, giving a value of the same relocatable type. Also, a value of some non-external relocatable type may have a value of the same type subtracted from it, and the result is an a signed value of absolute type giving the number of bytes separating them.

5. Statements

5.1. Expression Statement

A lone expression may form a statement, and causes a 16-bit word with that value, and with appropriate relocation bits, to be written in the code stream. The only restriction on the construction of such an expression is that it may not begin with a keyword (but one may parenthesise the keyword). The keywords are described below.

5.2. String Statement

A sequence of ASCII characters, surrounded by quotes ("") forms a sequence of 8-bit bytes containing the characters. The escapes allowed in character constants may also be used in strings.

5.3. Assignment Statement

If a name followed by an assignment sign ("="), followed by an expression, is written where a statement is expected, the assembler will set the value and type of the name to that of the expression. Any name may be redefined in this way, including the location counter symbol '.'. Assignments to '.' are restricted, in that they may not change its type or segment. Typical uses of the assignment statement with '.' are to reserve some amount of storage:

```
.=.+10
```

(which reserves ten words of storage), or to perform the function of an "origin" statement:

```
.=100^.
```

which causes assembly to commence at the hundredth word of the current segment. In the latter case, note that the use of the operator '^' is required, to

prevent the type of '.' from changing. Care must be taken when such statements appear in programmes which are to be later processed by the link editor.

5.4. Keyword Statements

Most statements begin with a keyword, as all the machine instructions, and most assembler pseudo-operations have the type "keyword". There are actually several keyword types, and the differing type values select the syntax expected for the operation by the assembler. Each pseudo-operation keyword has a value which identifies that particular pseudo-operation to the assembler. Machine operation keywords have a value equal to that of the machine op-code:

```
jst = 0x2000^jst
addb = 0x8000^addb
```

Only the values of the machine operations are guaranteed to withstand changes in the assembler implementation.

5.5. Operand Format

Instructions which take an operand all accept the same format, with the exception of the shift instructions (see below).

An address may have one of the following forms, where the expression may be of any type:

expr which causes the value of the expression to be placed in the address field of the instruction. The expression must evaluate to an even address, and must be expressible in 12 bits (the size of the address field).

$ *expr* defines a 16-bit literal with the value and type of *expr;* the address of the literal in the literal pool is placed in the address field of the instruction.

5.6. Machine Instructions

The names of a great many of these are different from those of the manufacturer's assembler; the old name is given in brackets following the new one. Certain instructions, particularly the branch-type instructions, may result in the generation of up to three hardware instructions. This is done to remove restrictions of the hardware, and to make coding easier; the programmer must use some care in rare cases where the actual size of instructions is important.

The instructions fall into five classes: skip instructions, simple instructions which take no operands, single operand instructions, shift instructions, and branch instructions.

5.6.1. Skip Instructions

These take no operands; the skip conditions are those provided by the hardware.

shz	**skpe**
skpk	**skpm**
skpfl	**skp**
sbne [sbne]	**sbeq** [slz]
sbge [slp]	**sblt** [slm]
skeq [skz]	**skge** [skp12]
sklt [skm]	

5.6.2. Simple Instructions

These perform various unary operations on the accumulator, or change the state of the processor in some way; they take no operands.

com [clone]	**inc**
neg [ctwo]	**dec** [deca]
hex	**halt**
refl	**refu**

5.6.3. Single-operand Instructions

These take the general format of address, as discussed above.

add [add12,ai]	**addb** [add]
ld [ldd,li]	**ldb** [load]
st [sta]	**stb** [sto]
jmp (=**br**)	**jst**
isk	**ioc**

Both the **add** and **ld** instructions actually have variants which allow immediate operands; the assembler generates the immediate form of instruction when the operand of either is a literal.

5.6.4. Shift Instructions

These take one operand, which must be a literal, in which the expression is of absolute type. The value of the expression must be no greater than 16.

als	**ars** [srt]
rol	**ror**

The hardware actually restricts the number of bits shifted to no more than 8, but the assembler arranges to generate an extra shift instruction, if that is required.

5.6.5. Branch Instructions

The assembler provides a set of conditional branch instructions, which compile into one or more skips (forming the opposite condition to that of the branch) over a jump instruction. The use of these instructions, rather than skips, tends to make programmes both easier to write, and easier to read. The set of conditions provided is not complete, so several branch instructions may be required.

bge	**bbge**
blt	**bblt**
bne	**bbne**
beq	**bbeq**

These instructions branch on: greater or equal to zero, less than zero, not equal to zero, and equal to zero, with tests provided for both words, and bytes.

5.7. Pseudo-operations

5.7.1. Location counter alignment: .even

The pseudo-operation **.even** ensures that the location counter '.' is on an even byte boundary, by writing a zero byte, if necessary.

5.7.2. Segment Switching: .text, .data, .bss

To switch from segment to segment, use:

.text
.data
.bss

as required.

5.7.3. Byte Values: .byte

The pseudo-operation **.byte** is followed by a sequence of expressions, whose values are assembled into successive 8-bit bytes:

.byte *expression* , ...

Each *expression* must be of absolute type, and may contain no external references.

5.7.4. External definitions and references: .globl

For each of the types: relocatable, undefined, and absolute, there is a corresponding type of external. If a symbol has the type undefined-external, it is expected to be defined in another file, and any references to it resolved by the link editor. If a symbol is of any other external type, it indicates that it is both known within this assembly, and may also be referenced by other files.

The pseudo-operation

.globl *name*, ...

causes the external attribute to be added to the type of the listed names.

5.7.5. Labelled Common: .comm

The statement

.comm *name*,*expression*

causes the *name*, if not otherwise defined, to acquire the type undefined-external with value that of the *expression* (which must be of absolute type). If the symbol is not redefined in another file, the link editor will place *name* in the bss segment,

before all symbols explicitly declared as such, leaving *expression* bytes unallocated following it.

5.7.6. Conditional assembly: .if and .endif

The assembler provides no macro instructions, but it does provide a simple form of conditional assembly. The statement

.if *expression*

will cause the statements following to be assembled if the expression, which must be of absolute type, and be defined in Pass 1, has a non-zero value. Should the value of the expression be zero, then statements following, up to the pseudo-operation

.endif

will be ignored, although all symbols appearing inside the **.if-.endif** statement set will be entered into the symbol table. The **.if-.endif** sets may be nested.

5.7.7. Specify Location of Literal Pool: .litorg

The assembler maintains a pool of the literals used in the address field of certain instructions, and automatically writes the pool, as addressibility considerations demand, or at the end of the assembly (when the remaining literals are written to the text segment). It is sometimes necessary to cause the literal table to be written prematurely, in order that a particular section of code remain uninterrupted (eg, a table). The programmer may force the contents of the literal table to be written, at any time, with the statement:

.litorg

On the Photon, the **.litorg** is provided mainly as a convenience, as the assembler only writes the literal pool once, at the end of the assembly.

6. Diagnostics

When an input file cannot be read, its name followed by a question mark is typed and assembly ceases. When syntactic or semantic errors occur, a single-character diagnostic is typed out together with the line number and the file name in which it occurred. Errors in pass 1 cause cancellation of pass 2. The possible errors are:

)	Parentheses error
>	Symbol table overflow
]	Parentheses error
"	String not terminated properly
.	Illegal assignment to '.'
A	Error in address
B	Operand too remote
E	Error in expression
F	Error in local ('f' or 'b') type symbol
G	Garbage (unknown) character

I	End of file inside an if
M	Multiply defined symbol as label
O	Word quantity assembled at odd address
P	'.' different in pass 1 and 2
R	Relocation error
U	Undefined symbol
X	Syntax error

7. References

1. Ritchie, D.M., "The UNIX Assembler Language", *Documents for Use with the UNIX Timesharing System*, available to UNIX licencees.
2. *The UNIX Programmer's Manual*, 6th Edition, Bell Laboratories Inc.

8. Appendix A: Sample Photon Programme

The following sample programme illustrates some common constructs of the language.

```
/ photon programme

.=0x100^.           / start at hex 100

.globl              x                   / make x known externally
.globl              z                   / refer to external z
.comm               q,6                 / labelled common q
1:
                    ldb                 x                   / load 8-bit x
                    neg
                    add                 $20                 / generates add-immediate
                    bge                 0f                  / special branch instruction
                    bbeq                0f
                    jmp                 .                   / self-referencing
0:
                    ld                  $y                  / relocatable literal
                    beq                 0f
                    jmp                 0b
0:
                    jmp                 1b

.data
str:
                    "hi there\n"        / a string
bytes:
                    .byte               3, 4
                    .byte               5
.even
y:                  x                   / relocatable expression
                    2
                    x+2
                    z+6                 / offset from external
                    2*[2+3]             / a more complicated expression
.bss
x: .=.+2
```