To: Distribution

From: T. H. Van Vleck

Date: April 10, 1981

Subject: Hexadecimal Floating Point

SUMMARY

This document describes how to support Hexadecimal Floating Point (HFP) in Multics. The best way to provide this support is:

1. Support HFP arithmetic in FORTRAN only.

2. Have all floating arithmetic in a (separately compiled) program be the same mode.

3. Modify system runtime routines to support I/O and debugging.

To provide this level of HFP support properly will involve substantial effort. The resulting system will provide many opportunities for user errors which lead to garbage results; some but not all of these errors can be checked for.

4. If HFP programs are run on a non-HFP CPU, the programs might appear to work but generate garbage answers. Detect this situation and cause an error stop.

5. Attempts to mix HFP and non-HFP programs in the same process may lead to errors in the interpretation of numbers. Again, the user's job will appear to work but the results will be wrong. The ability to mix modes, though, may be needed by sophisticated system builders and by programmers converting data from one mode to the other. Do not forbid mixing or even attempt to detect all cases.

6. Mixing HFP data and non-HFP programs, or vice versa, will also lead to garbage answers. Do not attempt to detect or prevent this problem.

These limitations will lead to problems and complaints and will increase the developers' support workload.

TECHNICAL ISSUES

Need for HFP

The need for HFP is discovered when comparing Multics language implementations with those of other vendors. Sometimes this comes about when a site attempts to convert programs from other machines; other times that the need for HFP is noted are when benchmarks or system proposals are being prepared by Marketing.

The Multics machine currently provides single and double precision binary floating point with a maximum exponent of 2**38. IBM machines support exponents of up to 2**75. If an otherwise valid program is converted from an IBM machine to Multics, it may encounter overflow or underflow conditions on Multics which it did not see on the IBM hardware.

The situation with other vendors' hardware is similar.

User sites have requested that we support HFP by SCP and RPQ.

Functions Required for HFP Support

Supporting HFP on Multics requires providing these functions:

- o Program compilation
- o Execution
- o Debugging

Furthermore, these functions must be provided in a way which is consistent with the rest of Multics facilities. Users have been led to expect a high degree of uniformity and consistency in system interfaces, and wish to be able to combine previously written programs, new programs, and system utilities to build complex subsystems without encountering implementation restrictions. For example, forcing the user to use a special debuggger for HFP instead of the standard probe command would be seen as an unreasonable limitation.

If we do an incomplete job of HFP support, we will regret it later: users will discover the deficiencies in our support sooner or later, perhaps at times inconvenient to them, and will request or demand that we complete the job, with much ill feeling on both sides.

The worst possible failure mode is one in which the system appears to accept the user's commands, but produces incorrect results without any indication of error. Even if these incorrect results are caused by a user mistake, properly warned against in some manual or info segment, the user can be expected to request that we "fix the system" so that the problem will be detectd automatically.

Hardware Implementation of HFP

The DPS8/70M and ORION processors support HFP arithmetic as described below. This feature was first designed for CP-6, for XDS compatibility; some design choices were made so it could be retrofitted into Current Product Line (CPL) processors.

DATA REPRESENTATION

HFP data is stored in the same number of words as Binary Floating Point (BFP) data, and the division of the word into exponent and mantissa is the same. However, the exponent for an HFP number is a power of 16 instead of 2, and the mantissa is therefore not always normalized to have a 1-bit in bit 9: instead, a normalized mantissa has its leftmost 1-bit somewhere in bits 9-12. This change loses a few bits of precision, compared to BFP, depending on the value of the exponent. (Called "wobbling precision" by numerical analysts.)

AMBIGUOUS OPCODES

There are no new operation codes assigned to cause the CPU to perform HFP operations on HFP data. Instead, the old BFP opcodes are used, and mean either BFP or HFP operations depending on the state of a CPU indicator. This opens the possibility of this indicator assuming a value which is incorrect for the data being operated on, either because the indicator gets set wrong, or because the data is not of the type appropriate.

NEW INDICATOR

The indicator value which selects whether BFP or HFP arithmetic will be performed is part of the Indicator register of the CPU. The <u>HFP flag</u> is bit 32 of the Indicator register. User programs can change the state of the HFP flag by using the LDI and RET instructions.

CPU MODE REGISTER

A new bit in the CPU mode register controls whether the Indicator register specification of HFP will be honored or not. If this bit is zero, the CPU will never do anything in HFP mode, regardless of what the user program specifies. If the bit is one, then the CPU will switch between BFP and HFP modes at user request.

Possible Scopes of HFP

This section describes implementation choices available to us in providing HFP support.

ONE LANGUAGE OR SEVERAL

It is not necessary to add HFP support to every Multics programming language. The need for HFP is presumably greatest for FORTRAN programs, since this is where scientific calculations involving floating point are most common. PL/I support of some kind might be necessary in order to deal sensibly with FORTRAN variables in HFP encoding, since the FORTRAN compiler, debuggers, and runtime support are written in PL/I. We have had problems in the past with conversion of APL programs from other systems, so HFP support for APL should be considered too.

PER RELEASE

One way to provide HFP support is to decide that all floating point data in Multics is now HFP and must be operated on by HFP instructions. This change could be made as part of a particular release of Multics; all software which had to know of the difference would be installed simultaneously at the site in a massive flag day, all programs using floating point would be recompiled, and all old BFP data would be converted in place to HFP format. (This was the approach taken by CP-6. They had no conversion problem because they never used BFP.)

Since not all Multics sites run the same release, we would have to invent conversion procedures for the export and import of floating point data and programs between sites. (Programs need conversion even though the opcodes are the same, because constants in the programs will have different encodings in BFP and HFP.)

Even worse, some sites will be running Multics on CPL hardware, which does not support HFP. If such a site obtains a program which depends on HFP, and tries to run it, it will silently produce wrong answers. The user program can switch the HFP flag on and off all it wants, but all numbers will be interpreted as BFP; in particular, program constants will be given the wrong interpretation. This is unacceptable behavior. To prevent it, the simplest way is to modify the system runtime for all languages to check whether a program being invoked uses HFP, and arrange that compilers producing object segments mark those that need the HFP feature. A flag in, say, wired_hardcore_data indicating the presence (and enabling) of the HFP feature would then be checked at every program invocation. (This check imposes a performance penalty on every program in the system.) If a program which requires HFP is executed on a non-HFP CPU, an error condition is signalled.

The file conversion programs mentioned above will be extremely difficult to construct. Floating point numbers are indistinguishable from other bit strings when stored in a file; in general, only the programmer of the application program which created a file can produce the program which converts the floating point numbers in the file. The file conversion programs must be restartable, in case there is an interruption of some kind while a file is being converted; and they must be kept around forever, since data files too will be traded between sites and retrieved from dump tapes. Worst of all, we have no standard place to indicate that a file contains HFP data, so no standard check against data misinterpretation can be introduced.

PER SITE

A variation of the per-release scheme allows a site to choose whether to have BFP or HFP data at release change time. Communication between sites still requires knowing whether conversion is required and might require conversion programs. But sites which did not choose to use HFP would not have to go through the massive data conversion in order to put up the release with HFP support.

Both the per-release and per-site approaches minimize the possibility of ambiguous data; if a number is interpreted as floating point, the correct value for the HFP flag is well known, and is a constant at the site. But the possibility of transmitting HFP data and programs to a non-HFP site is present in this scheme as well; therefore, we need to detect mismatches between desired and supplied encoding, and if there is a mismatch, we need file conversion programs.

Allowing more than one kind of floating point number involves Multics Development in dual maintenance of one form or another. There will be parts of the system used only in HFP, and others only for BFP, and these parts will both require checkout and maintenance. If the per-site option is chosen, separate checkout systems will be needed for maintenance.

PER PROCESS

If some processes wish to do BFP arithmetic and others HFP, then the HFP flag must be set correctly for each process, and the system must not pass the flag inadvertently from one process to another. The supervisor uses floating point arithmetic in a few places itself, so in fact we must implement per-ring management of the HFP flag to prevent an outer ring from interfering with an inner ring's calculations.

If all programs in a process are supposed to be one flavor or another, we can provide consistent I/O routines and compilers by the search rule mechanism, but we still must check to prevent the introduction of a subroutine of the wrong flavor into a process, because the search rule mechanism is often the source of user confusion.

The file conversion problem still exists in this case. We need ways to discover whether conversion is needed, and means to do the conversion. If a user wishes to combine some subsystems using BFP and some using HFP in the same process, or wishes to read some data files containing BFP data and other data files containing HFP data, he encounters severe problems.

The per-process approach is the one chosen by GCOS. They support FORTRAN HFP compilation and execution, and produce an error message at runtime if an attempt is made to combine HFP and BFP object units into a core image. The FORTRAN code manipulates bit 32 directly, saving and restoring it around external calls; user subsystems may call special routines to convert numbers and to manipulate the HFP flag.

If we choose the per-process option, a site could use the Access Isolation Mechanism (AIM) to separate HFP programs from BFP, by placing all users in either the "binary" or "hex" compartments. A user from one compartment is prevented from reading data created in the other without the intervention of the system security officer. Unfortunately, this compartmentalization is very strong; it includes all data, not just floating point numbers, so that the compartments are unable to communicate by mail, for example.

PER PROGRAM

The next most general situation is one in which an individual program chooses HFP or BFP operations for all floating point variables in the program. This approach allows the programmer to choose the type of data representation most appropriate for the calculation being performed, and assures that the operation of a program is not interfered with by the choice of environment it is run in (since all Multics processes would be alike in their ability to run either HFP or BFP). This proposal treats HFP data as simply one more data type, as different from BFP as integer is different from floating point.

In order to provide this level of implementation, the HFP flag changes state dynamically in a process depending on the intentions of the compiled code. Since the HFP flag can be changed by slave mode instructions, this convention is fairly easy to implement. A management convention needs to be defined which will be observed by all programs: one possibility is to set this flag to its required value whenever its state was unknown, or known to be incorrect.

The current Multics PL/I call operator sets the HFP flag to zero when a called procedure returns. Multics convention is to save and restore indicators across a call, but PL/I knows that indicators=0 will be just as correct after a call and loads with zero as an optimization. The same is true for FORTRAN.

Conversion of whole packages to Multics from other vendors' systems is straightforward under this scheme; but combining subsystems still raises the possibility of attempting to combine HFP and BFP programs in the same process. Communication between such programs via file is no different from the situations already discussed; but now we face the additional possibility of communication across calls. If all data in a program is either HFP or BFP, there is no way to write a valid program which takes one kind of variable as argument and does the other flavor of arithmetic, or passes the other flavor as argument to a subprogram. Through carelessness or misunderstanding, however, users may mismatch arguments across a call; and Multics does not now check parameter matching across calls. (MTB-094 describes how runtime parameter checking could be implemented. HFP would make it even more desirable.)

PER VARIABLE

The most general approach is to allow the programmer to choose the data representation for individual variables within a program. A single program can then perform either HFP or BFP arithmetic as necessary, according to declarations under control of the programmer. Implementation of this level of generality requires no additional runtime complexity over the per-program scheme, except that the individual language compilers will need modification to permit the expression of the programmer's wishes. It is possible that we may choose to provide per-program HFP support for some compilers, per-variable in others, and only BFP in still others.

Per-variable support allows the programmer to create a straightforward program which adapts HFP and BFP environments to each other. Without this support, a user cannot write a file conversion program without calling on some external subroutine.

Per-variable support represents a non-standard language extension. Sites which wished to encourage the use of standard language features only, such as Avon, would find difficulty in exporting Multics programs which used this special feature to any other machine.

CONCLUSIONS: SCOPES OF HFP

Given the advantages and disadvantages of each possible scope of implementation, it seems best to choose the per-program scope. Per-release and per-site are ruled out by maintenance issues. Per-variable is too much work inside the compilers. Per-program is about as much work as per-process and fits more naturally with the rest of the Multics system.

Compiler Changes

This section discusses the changes necessary to each language if HFP support is desired for that language. If we decide not to support HFP for a particular language, we may still need to make some modifications to the runtime, or perhaps even to the compiler, in order to ensure that programs continue to run, such as resetting the HFP flag.

FORTRAN

Declaration of HFP Variables

If per-program support of HFP is chosen, the FORTRAN programmer needs a way of expressing his intention for the arithmetic to be performed by his program. A compiler control argument and a %options directive are likely to be desired to permit specification that a whole subroutine operate in HFP.

If per-variable support is desired, syntax extensions to the FORTRAN language to distinguish between BFP and HFP will be needed, and intrinsic functions to convert between the two representations will also be required.

Constants

Floating point numbers in FORTRAN programs have to be converted to HFP for use in HFP arithmetic. To provide this facility, the compiler front-end must know that the constant should be stored in HFP format, and have a conversion program which it can call to produce this constant. The listing generator probably needs the inverse function. Since the compiler is written in PL/I, these conversion programs must be available in PL/I. The current FORTRAN compiler performs this conversion inline, by converting a fixed decimal value to a floating value; to support HFP, we must either recode this and use per-variable HFP support in PL/I, or call an external routine.

Currently, the data type of a constant is obvious from its syntax; it may be necessary or desirable to invent a FORTRAN representation for HFP constants if per-variable support is chosen.

Compile-Time Arithmetic

The PARAMETER statement does its compile-time arithmetic by calling special FORTRAN routines to perform the interpretation of individual operators. Performing these operations in HFP is a matter of creating a additional set of HFP subroutines, if per-program support is chosen. Per-variable support would require additional complexity because of mixed mode operations.

Constant folding done as part of compile-time optimization is now done by in-line PL/I code. To do this in HFP requires per-variable HFP support in PL/I or recoding of this part of the compiler to use subroutines.

Precision of Intermediate Results

If per-variable HFP support is chosen, the compiler must choose how to compile mixed-mode arithmetic. It is not clear how we could discover whether the user would prefer additional precision or a bigger exponent, since this choice is data dependent; so the compiler must make an arbitrary choice.

Code Generation

The object segment must be marked in some way to indicate that it contains HFP operations, no matter what strategy is chosen. (One way to do this is to use new entry and call operators which check for the presence of the HFP feature and flag the stack frame so that runtime routines know HFP is being used.) In any case, the code generator may be called upon to indicate HFP object segments, argument descriptors, or input parameter lists.

If per-variable support is chosen, the code generator must manage the HFP flag as part of the machine state, and must be able to perform type conversions between BFP and HFP.

I/O Package Changes

The FORTRAN format conversion routines need to know what kind of data encoding they are working with. Current I/O statements pass a few bits to fortran_io_ describing what kind of storage values are being manipulated; this field could be extended to flag HFP, or a per-stack-frame flag could be set at program entry.

PL/I

Declaration of HFP Variables

If per-program support of HFP is chosen, the PL/I programmer needs a way of expressing his intention for the arithmetic to be performed by his program. A new option on the procedure statement and a new compiler control argument are the right way to do this.

Per-variable support requires extension of the PL/I language with a new attribute, orthogonal to base, scale, mode, and precision. This attribute has to be supported in all parts of the language, and builtin functions created to make conversion explicit.

Intermediate Results

Per-variable support requires the same sort of choice as was made for FORTRAN, although the semantics of the implementation would be made explicit, as is now done for the precision of intermediate results.

Code Generation

Because of the complexity of the PL/I language, any change which affects code generation probably cannot be added to the current compiler. Per-variable support falls into this category. Less complex changes, such as per-program support, can probably be done.

I/O Package Changes

The changes here are similar in magnitude to those for FORTRAN.

APL

Prospective customers trying out our APL have encountered problems with the smaller exponent provided by BFP. APL currently stores all floating values internally as double precision, so the best way to improve APL arithmetic is probably to change all numbers to HFP with some release. This could be done with an in-place workspace conversion invisible to the user;

minor problems might occur if users have PL/I external programs called by APL. Most of the work to convert APL to HFP involves modifying the I/O package and the operators. Since the APL interpreter is written in PL/I, it would be convenient to have HFP support in PL/I, but it is not strictly necessary. Paul Green points out that there may be some hidden problems with APL oerators which work on floating point, such as the matrix inversion operator, which may not be numerically stable over the expanded domain.

BASIC

We have had problems with precision in BASIC in the past, which led us to invent double precision BASIC. If we wish to improve the exponent range in BASIC, the situation will be like that for APL, except that "random numeric" files written by user programs will need conversion; the situation is like that of FORTRAN in that these files have no type indicators.

CUSTOMER MAINTAINED COMPILERS

Several sites have produced their own compilers for languages not provided by Honeywell: there are several versions of PASCAL, an ALGOL-68, a SNOBOL-4, and probably many others. Our HFP support strategy should not break these compilers or their generated code. Furthermore, some of these compilers may wish to support HFP in a manner compatible with the solution we choose for Honeywell software; so the standard we choose should be extensible to other compilers.

CONCLUSIONS: COMPILER CHANGES

We should begin by supporting HFP in FORTRAN only. This is the place where most trouble arises on conversions and benchmarks. Per-program support in FORTRAN can be done without HFP support in PL/I, although it may be somewhat awkward and inefficient.

HFP support in PL/I is a significant job, even for per-program scope. Any attempt to do this should be deferred until Version 3 PL/I.

BASIC and APL support should be deferred until justified.

Runtime Changes

NEW DATA TYPE DESCRIPTOR

The correct way to manage the differences between HFP and BFP is to assign a new data type code for argument descriptors which will indicate which type of floating point number is being passed as an argument, stored in a structure, and so forth, as we now do for binary versus decimal and similar distinctions.

Assigning the Descriptor

Actually we need 4 descriptor types, for the following data types:

- o real floating binary short HFP
- o real floating binary long HFP
- o complex floating binary short HFP
- o complex floating binary long HFP

paralleling the BFP values. This is a minor problem since we are running out of descriptor type numbers: this situation will have to be faced sooner or later anyway, and can be tackled by assigning an escape value and using multi-word descriptors.

Changes to assign_

Once HFP data can be described in an argument list, the system runtime routine assign_ can be called upon to convert other values to and from HFP. The PL/I runtime program any_to_any_ does most of the work for assign_; modifications to this routine are extremely difficult. If PL/I programs are to contain HFP constants, this work must be done.

Changes to ioa_

The widely-used system I/O routines ioa_ and formline_ must have additional conversion code added to format HFP data for output. These routines will be directed by argument descriptors in the calling sequence when formatting output.

CHANGES TO PROBE

To support HFP, probe needs to be able to compare, print, and input HFP values. To know which values are HFP, probe must be able to determine the flavor of a value from the symbol table. This means that the symbol table utility stu_ must be able to distinguish HFP from BFP. If per-program support is chosen, it may also be desirable to be able to determine whether the variables in a stack frame are HFP, perhaps from a stack frame flag. The best way to support these needs is to support HFP as a set of data types in runtime routines like assign_. The actual changes to probe will be minor given this support.

CHANGES TO DEBUG

The debug command is not as cleanly implemented as probe, but the changes for HFP will not be major, given the stu_ support described above.

CHANGES TO BUILTIN FUNCTIONS

The mathematical runtime library must be carefully checked to ensure that accurate results are returned in HFP. Routines such as arctangent have to be defined over an extended domain; other routines must be checked for numerical stability under wobbling precision. CP-6 rewrote their entire math runtime when they instituted HFP; GCOS has recently modified theirs to work in both modes. (Our current math runtime is old and is thought to have less accuracy than the GCOS library.) If we are lucky, we will be able to adopt some or all of the CP-6 or GCOS runtime routines, but substantial work is still likely to be needed in adapting and verifying these programs.

CHANGES TO PROCESS STATE MANAGEMENT

If the HFP flag is considered part of the process's state, then specifications must be provided for when the flag is turned on and off. These will probably involve many routines. Possible changes to the entry operator have been mentioned above. We must also check that the fault and interrupt handling path does not unexpectedly kick a program into HFP: fim and ii must be checked, as well as signal_ and most of the ALM-coded system runtime routines. If the linker is to check that an object segment is compatible with the CPU type, then additional work must be done to make this path efficient.

CHANGES TO PROCESSOR MANAGEMENT

The same state management specifications must be developed for management of the processor state, so that the HFP flag is not passed from one processor to another inadvertently.

The flag which tells whether HFP is allowed at a site must be set correctly by system initialization. Hybrid systems using a CPL processor with a DPS8/70M processor cannot enable HFP, unless we want to invent some complicated software to set the required CPU for processes using HFP; this would have some performance impact.

CHANGES TO BINDER

The binder must be changed to generate correct object segment flags telling whether the object segment contains HFP code, based on the flags of the component object segments.

We may wish to change the binder to check for user errors. It is not necessarily an error to have HFP and BFP programs in the same bound segment, but the binder must be changed to warn the user or to check argument match for all calls between components. MTB-094 describes the parameter checking changes.

CONCLUSIONS: RUNTIME CHANGES

In order to preserve the consistency of the Multics programming environment and to continue to provide the standard services, quite a few changes are necessary. Even the bare minimum is a lot of work; additional highly desirable improvements may be deferred or skipped because of the additonal resources needed to implement them.

Application Programs

CHANGES TO MRDS

MRDS currently stores user data in several formats, and accepts most data formats in argument lists. The second facility comes naturally with assign_ conversion to HFP, but further extension to MRDS might be required to prevent underflow or overflow when storing floating point numbers in the user's database. Adding support to MRDS create_mrds_db to permit the declaration of "complex float binary HFP" or similar values would be a fair amount of work.

CHANGES TO GCOS SIMULATOR

Users may attempt to execute HFP programs within the GCOS simulator. To support this, the simulator's state management must be checked to ensure that the HFP flag got and kept the desired value; additional checks are needed to detect the attempt to run HFP on a non-HFP processor.

ARRAY PROCESSOR SUPPORT

Plans are currently being made to provide array processor support on Multics, interfacing with a special-purpose processor. This device presumably accepts only BFP at present. Consultation with the company that is providing the array processor is necessary to discover:

- o Whether the array procesor can accept HFP
- o Whether it can accept mixed mode input
- o How the desired mode is communicated
- o How the Multics interface to the AP must change
- o What features should be provided.

CHANGES TO USER APPLICATION SOFTWARE

There are three aspects to application software changes: first, old software must continue to be usable; second, it may wish to be able to work with new HFP programs; and third, it may wish to use HFP to provide additional exponent range.

The first aspect is our job; but every applications subsystem will probably have to be checked to make sure that we have done our job correctly. The second aspect, checking for continued correctness if run in HFP mode, is potentially extremely difficult; it requires expert numerical analysts to insure that the application software doesn't start churning out garbage. If a user recompiles his software in HFP mode and executes it, the answers will in general be slightly different, due to the loss of precision in the mantissa. In some cases, this loss of precision may introduce numerical instability, so that the result of the program becomes wildly wrong, or an algorithm fails to converge. The third aspect may involve additional analysis and rework of data formats and conversion packages. None of this work can be done mechanically.

Honeywell software which is affected by HFP includes:

- o Multics Graphics System

User software affected by HFP includes:

- o Consistent System (MIT, AFDSC)
- o IMSL (MIT, USGS)
- o SPSS (USL, USGS)
- o Harwell (MIT)
- o Linpack (MIT)
- o Conversion Packages (Marketing)
- o CPS (USGS)
- o SAS (USGS)
- o STATPAC (USGS)
- o MINITAB (USGS, Avon)

- CAM (USGS)
- GINO (Avon)
- ISIS (Avon)
- GLIM (Avon)

Developers and maintainers of each of these packages must be contacted to check their estimates for HFP support for each level.

CONCLUSIONS: APPLICATIONS

Conversion to HFP for current users will represent a significant cost. We may therefore expect that some customers will never convert, and that others will delay conversion for a long time. If we expect continued growth of the Multics PARC, then it is better to make HFP available soon, so that future customers may avoid conversion.