A system for generating application program packages for use on small computers can produce both questionnaire-tailored packages for individual users and standard packages for general distribution.

The Modular Application Customizing System

by R. D. Gordon

The development of inexpensive small computers during the 1970s has put advanced data processing capabilities within reach of thousands of small businesses. Many are first-time users with no particular computer expertise. They need software support of at least a moderate level of sophistication, yet their size does not warrant an in-house programming staff. In commercial enterprises, initial data processing needs are likely to be for standard applications such as payroll, accounts receivable, and accounts payable, and a great many software packages are available for such applications. Usually the designers of such packages, adopting a horizontal approach, try to make them general enough for a broad range of users in different industries.

When applications are tailored for a specific industry, however, with specialized functions and terminology, a vertical approach is desirable.[1] The vertical approach is appropriate for packages that include billing and inventory applications, since programs suited to one type of inventory are not likely to be completely satisfactory for another.

The advantage of the vertical package is that, because it is specific, it is more efficient and easier to use. Its disadvantage is that it applies to only a limited number of users. If a package is broadened in function to apply more widely, it becomes more complex. Although it would be possible to design an application that handled, say, pricing for both lumber and poultry, such an application might well be confusing to the people who used it. A mechanism for simplifying such a comprehensive and complex application is the subject of this paper.

Specifically, the paper describes the Modular Application Customizing System (MACS), a software system for generating basic accounting applications for small computers. Through 1979 IBM had released, for distribution in the United States, 17 application packages developed with MACS for System/32 and System/34. Several of the packages were produced simultaneously in versions for Canada, the United Kingdom, Australia, and Latin America. A typical package contains about 100 programs that handle four applications (for instance, billing, inventory, accounts receivable, and sales analysis) and related housekeeping functions, along with executable procedures and menus. The programs are written in RPG (the Report Program Generator),[2] the language commonly used for commercial applications on System/32 and System/34. The products are modular in that they can be distributed singly or in combination, as well as in a full set, so a user of one application can later add another. The term *customizing* indicates that the packages can be tailored in advance by means of a brief questionnaire.

MACS provides access to a pool of application data from which various packages can be configured for use on a particular target system. The programming language of the applications and the characteristics of the target system are aspects of a specific implementation of MACS, not of the overall concept. While MACS is a package development tool, it can also be a production vehicle for packages tailored for individual users. This paper discusses the current implementation of MACS for applications on System/32 and System/34. It covers the major elements of the system, explains the principles of function selection, and describes the customizing process.

background

MACS had its origins in the late 1960s in a questionnaire-driven special installation aid known as the Application Customizer Service.[3] The questionnaire permitted the user to include or omit functions, define field sizes, and lay out reports, within the context of a predetermined application design. The applications initially produced by that aid were intended for use on the smallest configuration of the IBM System/3, a card system with 8K bytes of core storage. The customized output consisted of printed program-design instructions but no actual code, and the user of the service was therefore expected to have a programmer available, or at least a programming trainee.

As the installation service evolved, applications for the disk and keyboard-console versions of System/3 also were offered, and the questionnaires were translated for use in France, Germany, Italy, and Japan. Subsequently, machine-readable program code was added to the printed documentation, and product variations were devised for specific industries. Publications were available to guide users in those industries in answering the questionnaires to produce the appropriate variation.

Experience with this earlier system indicated that an application product ought not to be tied to a single comprehensive questionnaire, or at least not one that the end user would have to deal with. For an application that contains a broad variety of functions, the questionnaire becomes cumbersome and usually includes many questions that do not apply to a particular industry. In addition, the number of combinations of answer sets, each of which defines a unique system, poses a significant problem for testing and for developing user education and publications. The way a given product is to be offered is another consideration, since questionnaire customizing is sometimes appropriate and sometimes not. In some cases, a product may be offered with one or more customized versions and several specialized, fixed versions. These are nontechnical considerations and are independent of whatever process is used to construct an application. The process should allow alternatives.

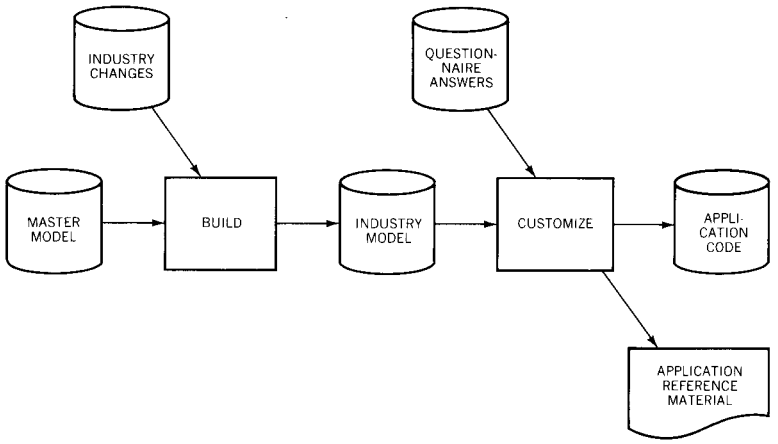
The problem addressed by the earlier customizer was entirely technical: how to generate a customized application. The task of releasing various versions of the same set of applications for different hardware configurations, in different countries, and under different business conditions identified a more general problem: how to derive several types of products from a common base. Ideally, such a base could provide applications not for just one target system, but for future systems as well.

As a practical matter, only some of an application's code is reusable, and a certain amount of turnover is essential. Parts of a package become obsolete and must be removed, a common result of moving the application from one target system to another. Improvements suggest themselves in some sections, either to correct deficiencies or to take advantage of new techniques. In addition, there is a continuing demand for new application function. Over the long term, the problem is one of evolution. Within the shorter term of a single product development cycle, the problem is one of flexibility and control. MACS was intended as a development vehicle that could be used in different ways and was itself amenable to change.

Components of MACS

The main components of MACS relate to the customizing run, the process that generates a set of applications from a questionnaire. If the product is offered in a customized form, the customizing run is a production run: each user fills out a questionnaire and later receives the printed and machine-readable output of the customization. If the product is offered in a fixed form, no questionnaire is used, and the customizing run is a development operation.

Figure 1 The building and customizing steps



Aside from answers to the questionnaire, if one is used, the input to the customizing run is known as the *industry model*. It contains the source material for the application product and the symbology that allows the source material to be customized. The processor of the industry model, called the *driver*, governs the format of the end product and to some extent its content. There are also numerous service routines which make up a *control program*. The control program acts as the host processor and provides the driver with various data management services, including the interface to the industry model. As part of the model interface, the control program interprets any customizing symbology and does the actual customizing.

master model

The industry model is created by a prior step which uses as input a *master model*, or master set of application material. This material is combined with a set of *industry changes*, selections, and overrides to convert the applications to a specific offering for a particular industry or country. The set of changes includes a definition of the questionnaire if the product is to be customized. The industry model is generated as a subset of the master model. This preliminary step is equivalent to answering a master questionnaire to customize the master model. This *building run* is a development process only, even if the product offering is to be customized. Figure 1 illustrates the building and customizing steps.

The control program of the customizing run and the processor of the building run are members of a set of service programs collectively known as the *support system*. The other functions of the support system are:

- Macro expansion, if macros have been used in the master model or the industry changes.

- File maintenance updating of the master model, the library of industry changes, and the macro library.
- Periodic reserialization of the master model and synchronization of industry changes to the new numbering.
- Tracking of translatable terms and text, to assist in translating the master model into languages other than English.

Division of the processor into support system and driver, like the separation of application data into master model and industry changes, is a simple but essential concept. The support system is intended to cover development housekeeping and functions that are unlikely to need change. The driver, on the other hand, is expected to change. It is designed to process an industry model of a given organization and produce certain types of output, some of which are in RPG, the source language of the target system or family of systems. Enhancements to the language imply changes to the driver, as do new types of devices attached to the target system, since new devices often require new types of output.

The customizing run is not used solely to generate the finished product. It is used also in developing application-related test programs, test files, educational aids, cross references, and statistics; and as new needs arise, they sometimes entail changes to the driver. The support system is viewed as static, and the driver as dynamic, with respect to long-term development, and their functions have been separated intentionally to simplify maintenance.

organizing principles

The master model and industry model are identical in organization and format. A set of industry changes is a supplement to the master model, providing a questionnaire, added application function, and reworded terminology. The set of changes can include overrides to individual lines or to large sections of text at any point in the master model. The primary purpose of the changes, however, is to identify those application functions from the master model that are to be included in the industry model, and those that are to be omitted.

Although each product might have its own set of industry changes and therefore its own industry model, that is not necessarily the case. A product offered in both fixed and customized versions has a single industry model, the fixed version being produced from a specific set of answers. If several closely related fixed products are to be offered, they could be produced from a single industry model with a development-only questionnaire and several sets of answers. Ordinarily there is a choice of methods, and the decision regarding which to use is based partly on the technical consideration of similarity and partly on such factors as product release scheduling and the organization and skill mix of the development group.

The master model holds the definition of an application, or of several different applications, based on a design for the target system. It is a partitioned data set, the sections of which generally fall into the following categories:

- Definition of the application data base: records and fields used in more than one program; fields internal to programs; files; algorithms for sizing fields and files; procedure sizings; data dictionary.
- Material that will become the machine-readable application code: RPG source code; sorting specifications; report and screen layouts; procedures; menus; system files.
- Source material for miscellaneous printed information: system descriptions; system flowcharts; operator run instructions.

A general philosophy for application development underlies the organizing principles of the master model. In the first place, most elements are defined only once. The application data base, for example, is intended for, but not restricted to, global terms. In practice, program-internal fields are defined as if they were global. This practice produces a consistency among programs that makes them easier to understand, and it reduces the opportunity for creating hazy definitions. It also eliminates most problems of duplicate maintenance.

A second principle is an intentional division of function. A program is considered to consist of three parts: its internal processing logic, its visible input and output (on screens and in printed reports), and its file input and output. This concept is close to the classic *input-process-output* concept, but not quite the same. The construction of the master model separates the three, and they are commonly assigned to different programmers. Aside from dividing the work, this construction leads to a certain uniformity in the product's visible input and output.

There are other benefits as well. One is that the application designer is forced to define the output in detail beforehand, instead of letting the programmer design it as he goes along. Another is that, since visible and file I/O usually are simpler than the program logic and can be completed earlier, it is possible to code them first and then use the customizing run to produce a skeleton program for the programmer to begin working with.

visibility

Another principle is visibility. The master model itself provides a librarian function which guarantees that all source material that will be in the finished product is in one place in a set organization, so it is fairly easy to find. The simple rule is that the current version of any code is the one in the master model, not one tucked in a programmer's desk drawer. An immediate benefit is that the

master model is a reliable reference source, not just for programmers but for application designers and publications writers. Therefore, in addition to formal code inspections, spot checks are carried out regularly. The intent to adapt existing applications to new systems implies that a good deal of the development effort will be devoted to redefinition and replacement. Fagan[4] has observed that small modifications, line for line, have a higher error frequency than wholly new modules. Making spot checking convenient helps reduce such errors.

Moreover, all updates must go through the support system, which provides a clear audit trail of changes to the master model. Each line of new and changed code in the current release level is date-stamped to identify the update listing for that change. The update listing shows a before-and-after image of the line. It can also show the initials of the person who made the change and a reference number for the documentation. Changes that are not current show the release level at which the last activity occurred; the last listing from that release level has a date stamp for the line. If need be, any line of code can be traced back to its origin.

The principle of visibility lies partly in the notation scheme employed. The programs in the application product will be in RPG; similarly, the master model notation for those programs is in RPG. In general, those sections of the master model that represent machine-readable outputs are expressed in the language of those outputs. The current implementation, in addition, permits alternate notations for many sections and provides a macro facility that allows macros to be defined and used anywhere. Operator run instructions, for instance, can be coded symbolically, written out verbatim, expressed in macros that expand into symbols or full text, or coded using these methods in combination. Although symbols and macros require prior planning, they are economical and generally consistent, but they are cryptic. Thus far it seems that nonprogrammers and master model programmers alike opt for a more obvious notation, even when it means writing more code.

Functions and switches

An application is an integrated set of data processing functions. For customizing, the set is divided into two categories: intrinsic and extrinsic. For example, the function list of an accounts receivable application could include the following:

- Keep track of money owed by each customer.
- Post purchases and payments.
- Print statements.
- Compute minimum payment.

- Check money owed against credit limit.

Of this list, the first two functions, and probably the third, ordinarily would be considered *intrinsic* to accounts receivable. The fourth function, minimum payment, applies to accounts receivable for, say, a department store. Since minimum payment is not part of every accounts receivable system, it is considered an *extrinsic* function. The fifth function, credit limit, is probably extrinsic because it is possible to have a workable accounts receivable application with no credit checking. On the other hand, an application designer might justifiably consider credit checking essential and arbitrarily treat that function as intrinsic.

characteristics

Three function characteristics of special significance are described as quantitative, restrictive, and modal. The first function in the above list—"keep track of money owed"—has two *quantitative* characteristics: the amount of money owed and the number of customers who owe money. A *restrictive* characteristic is illustrated by the third function—"print statements"—which can be expressed in the following mutually exclusive forms:

- Print statements for all customers.
- Print statements only for customers who owe money at month-end.
- Print statements only for customers who made a purchase this month.
- Print statements only for customers who made a purchase this month and still owe money at month-end.

A *modal* characteristic signifies how a function will be used or implemented, or both. The fifth function—"check money owed against credit limit"—implies the existence of customer credit limits. There are at least two ways of handling them: use standard credit limits and keep them in a table, or assign a unique credit limit to each customer and keep it in the customer's record. The former is less flexible, but it uses less storage. Considerations of storage space are of prime importance on small systems. The applications for System/32 were intended to provide satisfactory performance on a system with 16 000 bytes of main storage and 5 million bytes of disk storage. Whatever the size of the system, the input and output devices impose their own space requirements: screen dimensions, diskette sector length, number of card columns, printer line width.

To customize an application, MACS requires that the application's functions and function characteristics be expressed in terms of either switches or quantitative variables. A switch is an arbitrary number assigned to a particular function or characteristic. If on, the function or characteristic is present; if off, it is not. The entire master model is considered a text stream, each line of which is

conditioned by a switch or combination of switches. The switch settings indicate whether a given line is "true" or "false" for that customization. The customizer is a text selector: true lines are selected, false lines are bypassed.

use of switches

By and large, switches are used to indicate extrinsic functions and modal characteristics, and quantitative variables are used to describe quantitative characteristics. Intrinsic functions normally are not represented, since they are implied by some higher extrinsic function (such as the application itself), and any customization that included that extrinsic function would include all its intrinsic functions. Restrictive characteristics may or may not be represented, depending on whether the implementation of a function can be defined so that the user has the option of resolving the conflict either when installing the application (with the further option of changing it later) or when executing the job that performs the function.

The question of how to print statements, for example, can be answered by setting a code somewhere in the application. A tradeoff is involved in defining functions to permit restrictive flexibility, since whatever option the user does not elect implies a certain amount of dead code in the program. The presumption must be either that the implementation is so well conceived that the amount of dead code is minimal, or that the user will have the need to first make one choice and later make another.

model language

Control statements, written in what is known as *model language*, can be interspersed with the text of the model. The model language provides a vehicle for comments and allows the following basic operations:

- Define a numeric or character variable.
- Assign a value to a variable (by means of MOVE, ADD, SUBTRACT, MULTIPLY, or DIVIDE).
- Insert a variable into a line of text.
- Compare two variables.
- Set a switch on or off.
- Test a switch combination.
- Test a variable's characteristics (odd or even; actual length).

The *compare* and *test* operations are expressions that return true or false. Subsequent lines of text, or model language statements, can be conditioned on the truth or falsity of the previous *if* expression. An *if* expression itself can be so conditioned. The test-switch-combination statement provides for those few complex situations in which a line or block cannot be conditioned in a standard form and it is not worthwhile to define a separate switch to represent the combination.

The distinction between control program, driver, and model is significant in understanding the relationship of model text to switches and model language. A line in the model can be either text or a model language statement; a control character identifies the latter. Attached to each line is a switch condition. The control program resolves the switch condition and either accepts or bypasses the line according to whether the condition is true or false. If the line is a model language statement, the control program interprets and executes it; if text, the control program passes the line to the driver. The customizing logic represented by switch conditions and model language is thus external to the driver.

The customizing run

The customizing run has seven steps in three processing sections. Table 1 lists the steps, along with the nature of the model data needed. Each step produces some type of listing, which also is noted. Each of the outputs, whether printed or machine readable, is optional; special versions of the customizing run (to produce, say, spacing charts only) are commonly used during development. The first three steps involve the logical construction of the application or set of applications. At their conclusion, all of the application's inputs and outputs are determined and ready to be bound into the main product output, which will be largely machine readable.

Questionnaires

Step one resolves the questionnaire into the form needed for customizing, namely switch settings and the quantitative values that will be used to size the data base. The process depends on two industry model members, EQUATE and VALIDATE. EQUATE describes the questionnaire symbolically, allowing the driver to "equate" an answer to a switch or a variable. VALIDATE provides the rules for logical relationships: that mutually exclusive switches are not on together; that a secondary function is in fact accompanied by an appropriate higher-level function; that a monthly volume does not exceed its corresponding annual volume.

Even for a customized application product, only a few functions would be selected from a questionnaire. The presumption is that, since the product is intended for a particular industry, the persons in charge of defining the product can anticipate which functions are universally needed, which should be left out, and which require a question in the questionnaire. It might be that businesses in the industry are grouped into segments according to their product lines, and the questionnaire might ask which segment. The answer for each business, then, would determine an overall func-

Figure 2 Application-logic questions

1-06 Which basic pricing method do you prefer? Select one:

A. One price for each item.
B. Five prices for each item, selected by operator-keyed code. (Skip question 1-07 if this feature selected.)

1-07 If you wish to provide quantity discounts, how many quantity ranges do you want (optional feature)?

A. Three quantity ranges.
B. Five quantity ranges.

1-08 Do you want to include matching class discount/markup? Select one:

A. Yes.
B. No.

1-09 Standard discount/markup method. You must select one:

A. Code in customer record selects one of up to ten discount/markup percentages.
B. Code in customer record combined with item class code determines discount/markup percentage selected.
C. Code in customer record selects discount/markup percentage from four fields in each item record.

1-10 Additional pricing, taxing, and discounting options (mark as many as needed, or none).

A. Container charge. Calculates container charge for items whose records carry a unit container charge.
B. Local (county/city) sales taxes. Calculates and prints two local taxes in addition to the standard state sales tax.
C. Federal excise tax. Calculates and prints this tax, if it should be required, for selected items.
D. Trade discount on invoice total. Deducts a trade (as distinct from cash) discount from the preliminary invoice total, based either on a customer code or an amount.
E. Suggested retail price (SRP). Provides for a suggested retail price field in item records; this field may be used for printing SRP on invoices and price lists.
F. Pricing unit conversion. Calculates special prices for items on which the pricing unit is different from the inventory unit of measure (for example, wire inventoried by reel but priced by the foot).

tion mix. EQUATE, therefore, contains both unconditional switch settings for the universally needed functions, and conditional switch settings based on the questionnaire answers. For a fixed product, one without a questionnaire, all settings would be unconditional.

To date, the questionnaires used have been of the same general construction, with a section on application logic and sections on sizing of the data base. Application-logic questions, as illustrated in Figure 2, have a multiple-choice format.[5] Questions are numbered to indicate both section number and the number of the question within a section—for example, 1-06—and answers are indicated by letters. Answer A to question 1-06, then, is repre-

Figure 3 Digital questions

1-45 Maximum unit price for one item?

A. Less than one hundred dollars
B. Less than one thousand dollars
C. Less than ten thousand dollars
D. Less than 100 thousand dollars

1-46 Maximum extended price for one item?

A. Less than one thousand dollars
B. Less than ten thousand dollars
C. Less than 100 thousand dollars

1-47 Maximum amount of one invoice?

A. Less than one thousand dollars
B. Less than ten thousand dollars
C. Less than 100 thousand dollars

1-48 Maximum number of decimals—base price/cost?

A. Two (for example, $44.25)
B. Three ($22.125)
C. Four ($11.0625)

sented as 106A. Answers are required to some questions; other questions are optional. A question can be either conditional or unconditional; if conditional, it can be either answered or ignored, depending on the answer to a previous question. There may be more than one answer to some questions. Question 1-06 in Figure 2 requires an answer and is unconditional. Question 1-07 is optional and conditional. Question 1-10 allows for more than one answer.

quantitative variables

Questions used in sizing the data base are of three types: digital, numeric, and direct-size. Figure 3 illustrates some *digital* questions, which ask the number of digits necessary to accommodate key dollar and inventory volumes. Although the questions are in the same multipart form as the application logic questions, the answers are not used to set function switches, but are resolved into quantitative variables. In other words, marking "less than ten thousand dollars" is equivalent to saying "four digits." The dollar questions deal with whole dollars only. A separate question covers cents. The answers are subsequently combined to determine field lengths. The master model defines about two dozen quantitative variables for digital answers. It is EQUATE's responsibility to provide those answers, either through a questionnaire or by unconditional assignment. The relationships among an application's fields are such that these two dozen digital variables can yield appropriate and consistent sizes for some 500 fields.

Answers to questions on processing volumes provide data for *numeric* variables (see Figure 4). The numbers that are supplied are

Figure 4 Numeric questions

2-09	What is the average number of line items to be printed on an invoice?
2-10	What is the typical maximum number of line items to be printed on an invoice?
2-11	What is the maximum number of invoices and credit memos printed in a day?
2-12	What is the maximum number of invoices and credit memos printed in a month?

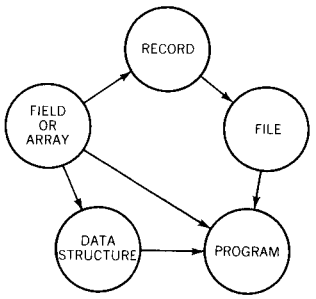
used to size files and determine disk space requirements. The difference between 100 and 500 invoices a day is of consequence even though both values are the same length.

Direct-size questions, the third type, constitute a catch-all category intended primarily for serial numbers and descriptions, which are data items whose lengths do not depend on user volumes.

Application data base

The second step of the customizing run is to define the terms in the data base and map their relationships to one another (see Figure 5). Each term has a label and a title. The label is the name used within the application code; the title is a phrase that describes the term for use in documentation. The label, the title, and a list of attributes constitute the definition of the term. The definitions are carried as text in the model and are selected by switches.

Figure 5 Data base mapping relationships



Fields local to programs are mapped directly to programs. Other fields are mapped to data structures or records. Just as a local program field can appear in more than one program, a field can occur in a number of data structures or records. Since most data structures are unique to a given program, however, data structures generally are defined locally within programs and not as part of the general data base. A data structure defined locally within a program need only refer by label to the fields it contains. The driver completes any missing part of the specification according to the global definition. A field's attributes may be locally redefined; the local redefinition can include the label as well.

Records are external collections of fields that are input to programs or output from them as a set; they are merely formats. They differ from data structures essentially in that records are associated with files and thus have the physical implication of an input or output device. A record can contain a data structure, in that a field can have subfields, and subfields can be mapped

within a field in a record; this was the only technique available prior to the introduction (with System/34) of the RPG data structure concept.

files

A file is a record-holding entity, the records having the same format or different formats. It has a width dimension, which must be sufficient to accommodate the longest record format. A file wholly contained on disk also has a capacity dimension, expressed as a record count. The two dimensions are converted by the driver to standard units, known as *segments*, for the device in question. The model defines the size of a segment. For System/32 and System/34, the segment is a 2560-byte block.

Files are included in the data base definition for either of two reasons. The first is that their record layouts are needed in program input and output specifications, with the specific exclusion of files that are primarily displays. The second is that they need physical disk space. All application disk files are included, since they meet the second criterion. Most meet the first as well, but not all. A work file used by a SORT program, for instance, is never used directly by an application program, but it uses disk space. A printer file is excluded because it is a display. A diskette file is excluded because it is both read and written by a utility, not an application program, and it uses no disk space. The disk image of that diskette file is included, however.

The mapping of records to a file establishes the file's record length in most cases. Exceptions are files with no defined records. For such files, the width is defined explicitly. Capacity is also defined explicitly where relevant. In both cases, the explicit definition is derived, through model language computations, from quantitative variables. Record length and capacity are two of a file's attributes; record padding is a third. For certain access methods, there are efficiencies in assigning a file a record length greater than it needs if the result is a length that is a submultiple of the disk's sector length. This relationship is expressed as

$$S/K - R = P$$

where S is the sector length, R the length of the file's longest record, P the amount of pad, and K the largest whole number that yields a positive whole number for P. S is supplied from the model, and R is determined from the file's record list. P is derived by the driver. Since the intent of padding is to increase a 63-byte record to 64 bytes, but not necessarily to similarly lengthen a 33-byte record, the model also supplies a pad allowance, A, expressed as a decimal fraction. The pad is accepted and added to the record length only if

$$P/(P + R) < A.$$

A fourth attribute is overhead space, expressed as extra segments

to be added to a file's total size (in effect, a capacity pad) or as the factors necessary to derive a file's index. The last attribute is a file-type code for files with miscellaneous characteristics that do not comfortably fall within the general width/capacity definition. An example is a record address file, whose records cannot cross sector boundaries.

procedure space

A procedure is a series of job steps, each of which involves the execution of a program. Programs that create or delete files use or relinquish disk space. A file is considered to be a permanent file, a work file, or an intermediate file. A permanent file, once created, is always present. A work file is created and deleted within the same procedure. An intermediate file is created by one procedure and deleted by another.

A procedure may or may not be able to reuse its own space. If not, its size is the sum of the sizes of its work files plus the sizes of the intermediate files that must be present when it executes. If it can reuse its own space, its size is the space required by its largest job step—that is, the sum of the sizes of the work files and intermediate files present at that point. For an application or set of applications to be usable on a given disk configuration, there must be sufficient disk capacity for execution of the largest procedure as well as for all permanent files and any other permanent disk occupants such as libraries and control files. This critical-point testing of disk space is the purpose of mapping files to procedures as part of the data base definition.

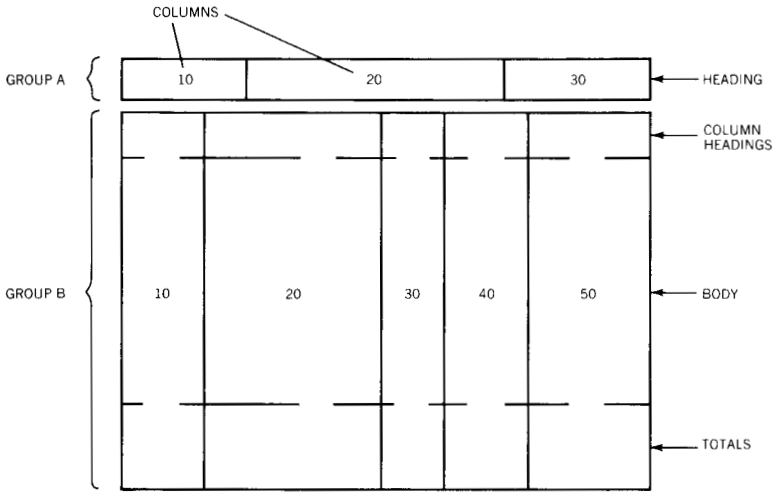
The data base step of the customizing run is the one in which conceptual changes to the driver and model are most likely as different target systems introduce advancements in the technique of data base management. The definitions of records and files used by MACS, for instance, are applicable at present but may not remain so. The MACS architecture is intended to be resilient enough to permit this type of evolution.

Reports and screen displays

The laying out of reports and screen displays, the third step of the customizing run, completes the logical construction of the application. A report or screen display is a visible output. A typical report has a heading which appears on every page, a body in columnar format, and column totals on the last page. In some cases, the heading-body-totals arrangement is nested. For instance, a sales report by region probably has an overall heading, a body that consists of a heading, body, and totals for each region, and then grand totals.

The purpose of the layout step is to assemble aesthetically acceptable visual outputs for any variation of a customized product. The model provides generalized layouts. The driver completes

Figure 6 Layout of a report in groups and columns



the layout by positioning all fields in a manner compatible with the sizing of the data base. For design purposes, an application's data base has a default size that is realistic for every field. The designer who lays out a report or screen display inevitably has in mind some probable set of switches that will be on, and as a result designs the original layout as a specific case.

The concept used in MACS for generalizing layouts is that the primary characteristic of any visual output is vertical alignment. A column of figures on a report is a list of numbers, usually with an explanatory heading, in vertical alignment. Similarly, a name-and-address block positioned to show through the opening of a window envelope exhibits vertical alignment. In MACS, such arrangements are called *columns*, and horizontal rows of columns are called *groups*. It turns out that a report or screen display can be described in its entirety in terms of groups and columns. A layout in the model is coded as RPG output specifications without actual output positions, but with a map to its groups and columns. For screen displays that double as input formats, the layout identifies the input fields so that the driver can derive the input layout as well.

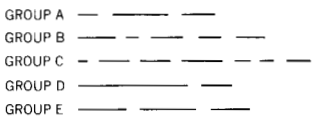
Figure 6 shows the layout of a report in groups and columns. A group consists of several lines, not all of which print at the same time. Group B in the illustration includes column headings, report body, and totals—three distinct output conditions. Each unique output condition in a group is called a *logical line*. An individual line image is known as a *physical line*. A double-line set of column headings is an example of a single logical line with two physical lines. MACS identifies a group by a letter, and a column by a

two-digit number; the notation *B20*, for example, refers to column 20 in group B. A field or constant occurs on a physical line. In lieu of an actual print position, it carries a reference that assigns it to a column. Multiple terms on a physical line can be assigned to the same column. Column B20 is thus represented by terms or sets of terms on several different physical lines. The widest set establishes the width of the column.

horizontal alignment

Within a column, a term can be displayed flush left, flush right, or centered. If there is only one term, the width of the term defines the width of the column. If the physical line has more than one term in a particular column, the outermost terms are aligned with the column edges, and the interior terms are aligned with the outer terms. Dummy terms are used to leave space between terms in a column. A column on a physical line can be double-defined—that is, defined with one set of terms and then redefined with another. That would be done, for instance, to print one or the other of different constants at the same point, each conditioned by its own output indicator.

Figure 7 One-dimensional layout



The concept of groups and columns allows the driver to determine all the column widths and then lay out the report or screen display without reference to the vertical dimension. Each group becomes a linear representation of column widths (see Figure 7). The map in the industry model supplies the guidelines for handling that representation. It gives any attributes for columns: space between columns, alignment of columns in different groups, specific print position. The widest group establishes the width of the report, just as the widest set of terms establishes the width of a column. A group can be shown flush left, flush right, or centered with respect to the widest group. A group can also be spread to the limits of the printer's width: the outermost columns are flush left and flush right with free space distributed equally between columns. If the report as a whole is narrower than the printer's width, the entire report can be set flush left, flush right, or centered with respect to the print margins.

If a group is too wide for the output device, the driver can define it (in the linear representation) either as self-contained in one line or as having a format that occupies as many lines as necessary. The multiline format subsequently is applied to the group's physical lines, as appropriate, converting any single lines to two or more. This solution is used in other report-generating processors, such as RPG's auto report function.

Columns of a self-contained group are disposed of one at a time until the group fits. This procedure is applicable when not all columns are essential. For this purpose, each column in a group can be assigned a priority class. The rightmost column in the lowest remaining priority class is dropped, and the driver continues de-

Table 1 Customizing run

Customizing step	*Model content*	*Printout*
Logical construction of application		
1. Resolve questionnaire and validate answers	• Questionnaire image • Syntax rules for answers • Relationships of answers to switches • Relational rules for switches and quantitative variables	• Summary of questions and answers
2. Construct data base	• List of fields, data structures, records, and files, and their attributes • Data base map	• Data dictionary • Contents of data structures, files, records • Analysis of disk space
3. Construct reports and screen displays	• Generalized layouts	• Spacing charts
Main product output		
4. Produce miscellaneous documents	• Text or symbology appropriate to document	• Operating instructions • Flowcharts
5. Produce machine-readable code	• Program calculation logic • Procedures	• Menus • Procedures • Program source code • Screen specifications
6. Produce machine-readable files	• Source data for files	• Contents of system files and test data
Reference material		
7. Produce cross-references		• Cross-references • Statistics

leting columns according to this rule until the group fits. For a report that contains both critical information and supportive information, the column-dropping strategy provides an aesthetic result where a group is too wide.

A major advantage of defining reports and screen displays as generalized layouts is that subsequent changes can be made inexpensively. It is often the case that a report looks different than it did on the original printer spacing chart and needs some minor changes. Or an enhancement to a program may call for new output fields. With a generalized layout, a field can be moved a space or two, or a new column can be inserted in the middle of the report, and the driver will adjust the entire format accordingly.

Main product output

Steps four, five, and six of the customizing run, as outlined in Table 1, are straightforward text-selection processes of the type described earlier. They differ from one another mostly in the details of format and handling, not in substance. Further, they are independent of one another, and the order of processing is arbitrary. Step four is for accessory documentation, either diagrams or text. It applies only to customized products. For a fixed product, the equivalent material would be supplied in a publication.

Step five produces the application source code: programs, control specifications for sorting, and the like. One of the outputs is a machine-readable directory for such material so that it can be arranged appropriately on the product's distribution medium, and so items can be duplicated if necessary. The transfer to the actual distribution medium is separate from the customizing run, to permit an intervening compilation step if object code is delivered. Step six produces machine-readable data files, either test data or control information needed for an application's housekeeping functions.

In each of these three steps, the driver binds the data base to the product output by incorporating record layout descriptions and field definitions into the printed data entry instructions; inserting record lengths, blocking factors, file sizes, and field lengths and positions throughout programs, control specifications, and procedures; formatting the records for the machine-readable data files; and attaching input and output specifications for records, screen displays, and reports to all programs. During this process the driver counts lines of code and makes up cross references to the main product output. The line-count statistics are helpful for development management. The cross references identify where every field was used across all application programs in the package.

Limitations

The customizing system has two principal limitations. The first is that it clearly is not interactive. The questionnaire and validation steps could be made interactive with minimal difficulty, but the rest of the customizing run is a serial process as presently conceived. There has been some work in this area, however. Current application packages produced through MACS include programs that allow a user to change answers to questions about processing volume and thereby revise the disk space requirements. Since the process does not involve recompiling of application programs, it is fairly rapid. The development of data base systems may permit a corresponding advance in the methods of sizing records, fields, and possibly reports.

The other limitation is that MACS requires no particular sequence of development and, as a result, provides no guidance for development. There is no single starting point, no section of the master model that must be done first when developing a new application. The reason is that MACS is not an application design aid. But application design is of course the first step, and the design must be complete and internally consistent, particularly with regard to switches and data base items, which will be defined globally. MACS is easiest to use in a development cycle that has definite design-review-coding stages.

Summary

MACS is a development tool intended to address the practical problem of mass producing standard small-business computer applications in a variety of configurations. This paper has described the current implementation of the tool, which consists of four elements: support system, driver, master model, and industry-specific changes. The support system consolidates master model and industry changes to form the industry model, which in turn is processed by the driver, using services of the support system, in a customizing run to produce tailored application packages. The driver, master model, industry changes, and customizing run are defined by the specific implementation; the one described here has been used for several years to produce System/32 and System/34 programming packages.

The thesis of MACS is that code is reusable, that the investment in design and programming of one application package can be retained as the initial capital for the next, so long as it is kept in an accessible form. An implementation of MACS is based on the way the driver and master model are conceived and designed, as well as on the intent of the customizing run. A formal definition of the system would be silent on those matters. The current implementation is straightforward, not revolutionary; the customizing run described here is essentially assembly followed by simple text selection.

One might wonder whether, for instance, a design language could be used instead of RPG notation for the application code itself. The language could be translated into RPG or, say, COBOL, since that would clearly expand the package potential of the master model. Indeed, such a language could be used. As a practical strategy, however, that approach would be premature since it would depend on development of the language, a better understanding of structured design than now exists, and an established working theory of how to apply the principles of structured programming to RPG. But it is not unlikely that all the pieces might fall into place for some future implementation.

CITED REFERENCES

1. L. B. Marienthal, "Selling small business systems," *Datamation* **24,** No. 10, 86–90 (October 1978).
2. *Introduction to RPG II*, IBM Systems Library, order number GC21-7514, available through IBM branch offices.
3. *Application Customizer Service System/3 Application Description*, IBM Systems Library, order number GH20-0628, available through IBM branch offices.
4. M. E. Fagan, "Design and code inspections to reduce errors in program development," *IBM Systems Journal* **15,** No. 3, 182–211 (1976).
5. Figure 2 reproduces part of the *Hardware Distributors Management Accounting System Questionnaire*, IBM Systems Library, order number GH30-0066, available through IBM branch offices.

The author's mailing address is IBM Corporation, General Systems Division, 2800 Sand Hill Road, Menlo Park, CA 94025.

Reprint Order No. G321-5136.